AFRL-SN-WP-TR-2002-1102

FILE PROFILING FOR INSIDER THREATS

Peter Reiher

Laboratory for Advanced Systems Research
Computer Science Department
University of California
Los Angeles, CA 90095

FEBRUARY 2002

FINAL REPORT FOR 06 NOVEMBER 2000 – 27 FEBRUARY 2002

SENSORS DIRECTORATE
AIR FORCE RESEARCH LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7318

NOTICE

THIS TECHNICAL REPORT HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION.

[illegible]
Martin R Stytz, Ph.D.
Project Engineer
Electronic Warfare Branch
Sensor Applications & Demonstrations Division

[illegible]
Charles M. Plant, Jr.
Branch Chief
Electronic Warfare Branch
Sensor Applications & Demonstrations Division

[illegible]
Paul J. Westcott
Division Chief
Sensor Applications & Demonstrations Division

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE ***(DD-MM-YY)*** February 2002	**2. REPORT TYPE** Final	**3. DATES COVERED** ***(From - To)*** 11/06/2000 – 02/27/2002

4. TITLE AND SUBTITLE FILE PROFILING FOR INSIDER THREATS	**5a. CONTRACT NUMBER** F33615-00-C-1746
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER 69199F
6. AUTHOR(S) Peter Reiher	**5d. PROJECT NUMBER** ARPS
	5e. TASK NUMBER NZ
	5f. WORK UNIT NUMBER 09
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Laboratory for Advanced Systems Research, Computer Science Department, University of California, Los Angeles, CA 90095	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) SENSORS DIRECTORATE, AIR FORCE RESEARCH LABORATORY, AIR FORCE MATERIEL COMMAND, WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7318	**10. SPONSORING/MONITORING AGENCY ACRONYM(S)** AFRL/SNZW
	11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S) AFRL-SN-WP-TR-2002-1102

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES
References are not defined on pages 16 and 17. Section numbering sequence is off throughout the document.

14. ABSTRACT ***(Maximum 200 Words)***
The goal of this project was to demonstrate that it is possible to detect insider misbehavior by careful examination of the file access characteristics of users. Our thesis was that ordinary user behavior would be characterized by types of file access behavior that were recognizably different than patterns exhibited by an insider attempting to gain improper privileges or making improper use of his existing privileges. As a proof-of-concept, we did not propose to build a deployable system, but instead to examine the idea carefully enough to determine if it was feasible. This research project has successfully demonstrated that a user's file access behavior can be analyzed to determine when he stops behaving properly and starts engaging in suspicious activity. The performance costs of gathering the data are acceptable, demonstrated by the fact that over a period of 2 years, we received no complaints about system slowness. (On previous projects where the experimental system was not sufficiently fast, our local users have never been reluctant to complain about the impact of testing on their work.) The amount of data we gathered was vast, but a real system need not keep all gathered data, and could probably reduce batches of traced records to model data frequently. Further, some of the data we traced has so far given us no advantage in detecting insider threats, so a real system would not need to gather this data. Demonstrating that our methods can be used in an experimental environment and actually making them work in a real environment are two different problems. We have only addressed the first. More research and development would be necessary to deal with the second.

15. SUBJECT TERMS
user profiling, file activity profiling, information assurance

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT:**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** (Monitor)
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**	SAR	70	Martin R. Stytz
Unclassified	Unclassified	Unclassified			**19b. TELEPHONE NUMBER** *(Include Area Code)* (937) 255-2811 x4380

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39-18

TABLE OF CONTENTS

SECTION NUMBER	TITLE	PAGE
1	Introduction	1
2	Research Approach	3
3	Trace Gathering	4
4	Model Building	6
5	Performance Against Plan and Future Work	16
6	Conclusions	26
References		27
Appendix A		28
Appendix B		31

File Profiling for Insider Threats

ABSTRACT/EXECUTIVE SUMMARY

Insiders pose special threats to computer systems. An insider is defined as someone who has some level of legitimate access to a computer system. An insider thus has a foothold in the system, and will normally be performing some activity there. Unlike an intruder, who must first obtain access, an insider typically focuses directly on doing damage or gaining unauthorized privileges. Thus, intrusion detection systems are often not properly designed to meet the threats posed by insiders. In the most recent CSI/FBI Report on Computer Security Issues and Trends [Power02], 65% of responding organizations reported one or more incidents of insider abuse that led to financial losses. The average loss reported here at UCLA and due to an insider attack was $300,000, with total losses for all reporting agencies of over $4 million.

Insider threats are commonly regarded as among the most difficult to combat [Schneier 00a]. Firewalls and other perimeter defenses will not work, as the insider is already inside them. Any attempt to base security on the secrecy of your procedures will fail, since the insider properly knows many such procedures and is in a particularly good position to find out about any he does not yet know. To some extent, possibly a great extent, he is trusted. The insider knows the weaknesses of the system, and can thus go directly for the throat. In some cases, the insider may have had a hand in designing the system. Detecting insider threats is commonly regarded as an unsolved security problem.

The goal of this project was to demonstrate that it is possible to detect insider misbehavior by careful examination of the file access characteristics of users. For modern computer systems, practically nothing of any significance can be done without accessing files. Running programs, viewing data, accessing networks and other devices, performing standard housekeeping functions, even determining the basic configuration of the machine and operating system, all rely on opening, reading, and writing files. Unlike other fundamental computer events, like keystrokes or mouse movements, the file system also works at a relatively high semantic level. Thus, file activity is both ubiquitous and relatively semantically meaningful, making it a promising arena to search for traces of what users are doing. If they are misbehaving, perhaps these traces can allow that misbehavior to be recognized quickly.

This research project successfully demonstrated that a user's file access behavior can be analyzed to determine when he stops behaving properly and starts engaging in suspicious activity. The performance costs of gathering the data are acceptable, demonstrated by the fact that over a period of two years we received no complaints related to system slowness. (On previous projects where the experimental system was not sufficiently fast, our local users have

never been reluctant to complain about the impact of testing on their work.) The amount of data we gathered was vast, but a real system need not keep all gathered data, and could probably reduce batches of traced records to model data frequently. Further, some of the data we traced has, so far, given us no advantage in detecting insider threats, so a real system would not need to gather this data.

Demonstrating that the method can be used in an experimental environment and actually making it work in a real environment are two different problems. We have only addressed the first. More research and development will be necessary to deal with the second. This project essentially suggests that this additional research and development would be worthwhile.

The obvious next step in this research is taking the very simple system that successfully detected a few attacks into an actual system that detects insider threats using this method, possibly integrated with other methods. To go beyond the already-completed proof-of-concept to this next useful step is a large amount of work, since the added value will be augmenting a real system with a facility that compares ongoing file system accesses to models of proper behavior.

File Profiling for Insider Threats

Final Report

File Profiling for Insider Threats
Final Report

1. Introduction

Insiders pose special threats to computer systems. An insider is defined as someone who has some level of legitimate access to a computer system. An insider thus has a foothold in the system, and will normally be performing some activity there. Unlike an intruder, who must first obtain access, an insider typically focuses directly on doing damage or gaining unauthorized privileges. Thus, intrusion detection systems are often not properly designed to meet the threats posed by insiders. In the most recent CSI/FBI Report on Computer Security Issues and Trends [Power02], 65% of responding organizations reported one or more incidents of insider abuse that led to financial losses. The average loss reported here at UCLA and due to an insider attack was $300,000, with total losses for all reporting agencies of over $4 million.

Insider threats are commonly regarded as among the most difficult to combat [Schneier 00a]. Firewalls and other perimeter defenses will not work, as the insider is already inside them. Any attempt to base security on the secrecy of your procedures will fail, since the insider properly knows many such procedures and is in a particularly good position to find out about any he does not yet know. To some extent, possibly a great extent, he is trusted. The insider knows the weaknesses of the system, and can thus go directly for the throat. In some cases, the insider may have had a hand in designing the system. Detecting insider threats is commonly regarded as an unsolved security problem.

The goal of this project was to demonstrate that it is possible to detect insider misbehavior by careful examination of the file access characteristics of users. For modern computer systems, practically nothing of any significance can be done without accessing files. Running programs, viewing data, accessing networks and other devices, performing standard housekeeping functions, even determining the basic configuration of the machine and operating system, all rely on opening, reading, and writing files. Unlike other fundamental computer events, like keystrokes or mouse movements, the file system also works at a relatively high semantic level.

Thus, file activity is both ubiquitous and relatively semantically meaningful, making it a promising arena to search for traces of what users are doing. If they are misbehaving, perhaps these traces can allow that misbehavior to be recognized quickly.

Our thesis was that ordinary user actions could be characterized by types of file access behavior that are recognizably different than the patterns exhibited by an insider attempting to gain improper privileges or making improper use of his existing privileges. Because insiders have usually worked on a machine for a significant period of time before they start to behave badly, we believed we could make models of their early good behavior that would allow us to determine, by observing deviations from the model, when they had gone bad. Further, information about the particular files being accessed might prove useful in distinguishing uninteresting variations in behavior from dangerous new patterns that could compromise the system.

As a proof-of-concept, we did not propose to build a deployable system, but instead to examine the idea carefully enough to determine if it was feasible.

This final report summarizes our findings from this project. We describe our research approach, the methods we used to gather data, our analysis of that data, and the experiments performed to demonstrate that models and statistics obtainable from that data could be used to differentiate proper and improper insider behaviors. We offer our conclusions and suggest possible future directions for research based on our findings.

Section 2 describes our basic approach, and Section 3 describes how we gathered our data and characterizes that data. Section 4 describes various models we derived from the data that allow us to detect many kinds of activities that are common when an insider starts to misbehave. Section 5 discusses our performance against our original plan and outlines future work. Research conclusions from this effort are provided in Section 6

2. Research Approach

Our experience with tracing file system activities for past projects suggested to us that such traces contain much useful information about what users are doing. Relatively little can be done in modern operating systems without making extensive use of the file system, particularly because the Unix paradigm of overloading the concept of a file to include devices, interprocess communications channels, and even processes themselves causes seemingly non-file operations to work through the file system. If everything a user does produces file system activity, perhaps traces of that activity can be analyzed to distinguish between good behavior and bad.

This approach seemed especially fruitful for detecting misbehavior by insiders. An intrusion detection system (IDS) is generally focused on finding attempts to break into system. Insiders are already inside the system, by definition, and thus their misbehavior may evade detection by an IDS. On the other hand, insiders have typically developed a characteristic pattern of behavior over the course of performing their proper duties, so the system can build a baseline of proper insider behavior to compare against ongoing behavior. If the deviation between the two is great enough, perhaps that is a signal of an insider threat.

This research thesis rested upon several unproven assumptions:

1. It assumes that file system activity is regular enough that we can make useful models of normal behavior. If normal user behavior results in highly varied patterns of file system access, we will be unable to differentiate between such random noise and actual attempts to misuse the system. Another aspect of this assumption is that user behavior is largely stable over long periods of time. If a day's activity is quite regular, but the next day's activity is equally regular in a different way, we will not be able to build a useful model over a reasonable period of time.

2. It assumes that attempts to behave improperly are distinguishable from normal behavior at the file system level. It is certainly conceivable that both proper and improper behavior result in very similar calls to the file system, with the actual damage being done through other mechanisms or deep inside the actual data provided to file system calls. Were this true, while we might be able to make decent models of normal behavior, insider misbehavior would also match those models, rendering the approach useless. While it seemed obvious to us that there would be many visible differences, it was certainly possible that setting the sensitivity of the system low enough to catch most insider threats would cause an unacceptable number of false positives from the system, a lesser (but still damaging) form of this problem.

3. It assumes that both models of normal behavior and checks of actual behavior against those models can be made fast enough to be of use. The approach would be most useful if it could check user behavior against models rapidly enough to detect problems in real time. At worst, it must be able to analyze traces of behavior against models rapidly enough to give system administrators signals of problems soon enough to counter the insider threat.

The major goal of our project was to develop enough evidence and understanding to answer these questions. We believe that we have done so, though we did not perform the exact steps outlined in the proposal, and we do not have a working implementation that detects problems in real time.

3. *Trace Gathering*

Our approach to this research was based on our existing expertise at gathering and analyzing file system traces. At first glance, trace gathering seems easy, and indeed, it is not an overwhelmingly challenging problem. But there are many details that make it hard to gather large amounts of trace data reliably, which is why it is so rarely done. Further, if one is not careful about how the data is gathered, analysis can become impossible. With the greatest of care, proper analysis of large amounts of trace data is still a daunting task.

Our first step was to gather a great deal of trace data. Of necessity, we gathered data from machines under our own control, which implied that we instrumented machines used by professors, graduate students, and other researchers. This necessity led to another challenge, since we now had to address the issue of whether lessons learned by close analysis of computer science researchers would be applicable to others who use computers. We address this challenge in the conclusions of the report.

Our laboratory uses Linux primarily for its day-to-day operations. Some work is done under Windows 95 and Windows NT, but the bulk of the computer use in our lab is done under Linux. Also, our instrumentation experience was largely in Linux. Thus, we gathered traces of Linux-based activity only, except to the extent that remote Samba accesses from Windows machines were served from an instrumented Linux machine. The instrumentation in question required minor patches to the operating system, since we needed to capture all file system calls, as well as a few system calls that were not directly related to the file system but proved to be necessary if we wanted to trace what was really going on in the system. (In total, around 35 system calls of various sorts were trapped and recorded.) We had performed such instrumentation for such DARPA-funded projects as Ficus and Travler, but keeping up to date with new versions of the Linux kernel and adding the extra functionality required for this project required further alterations to kernel tracing.

Our researchers use laptops for their primary machines. Sometimes these laptops are connected to the network, but often they are not. The file tracing solution thus needed to store traces on each machine's own disk, but because many of those disks had limited capacity, they also had to be transferred off the laptop disks to a more permanently reachable site on a server. Experience has shown us that this process must be fully automated or users on tracing machines will not accept it. Automating the process requires a modest amount of care to ensure proper behavior even when disks are full and time is limited. A server machine was also traced. Unlike the laptops, this server primarily provided web and mail service for a large number of users, rather than acting as a personal computer.

Each traced event was a call to the operating system, most frequently to perform some file system activity, but also for other purposes—such as creating a new process. The information stored from the traced event was typically the type of the call, the user making the call, timestamps, the identity of the file targeted by the call, and other header-like information related to the call. We did not trace data provided to or returned by file system calls. Doing so would have quickly multiplied the size of the real file system by several times. Further, realistic time pressures make it infeasible to analyze all data in all file system calls. Also, there are privacy considerations. While users in our lab relinquish some degree of privacy in the interests of research, a system based on an unreasonable weakening of privacy is not likely to become popular in the real world. None of our users ever complained about noticeable performance impact due to the tracing code. For typical user systems, we believe that this level of tracing can be done without disturbing theusers.

Ultimately, we traced the file system activities for eight machines representing ten real human users (and several daemon users) over the course of two years. These traces captured approximately 1.5 billion file system calls. These traces are still in existence and being used for this and other purposes. In Section 5 we discuss methods of sanitizing the traces, which would allow us to release them for use by others.

Each record in these traces represents a single system call, typically a file open, close, read, or write. Other file system calls (such as renaming a file, directory operations, and accessing or setting the metadata for a file) are also represented by records. Similarly, process creation and destruction system calls, and a few other system calls related to process management, produced trace records. Traces are organized into files, each representing the data gathered from a particular machine over a particular period of time.

Because of the vast volume of trace data, almost all trace analysis was done with automated tools. These tools typically read a file or set of files and search them for records of certain types. For example, the tool might allow counting of all file opens performed by a particular user, or it might make a list of the names of the files the user accessed. Where necessary, we performed some hand inspection of traces. In most cases, hand inspection was used to gain an understanding of the underlying behavior of particular programs, or to learn why counterintuitive results from the summarization tools occurred. We were careful to test that our trace analysis tools produced correct results before relying on them.

Our traces are organized as time-ordered logs, with entries made in the order they occurred on the machine in question. As a result, adjacent entries in the log do not necessarily represent a history of one thread of activity. For example, one log entry might represent a mail program running in the background checking to see if new mail has arrived, while the next log entry represents a foreground text editor opening a file. For some types of analysis, it is necessary to extract the set of log entries that represent a single thread of application file accesses. The process identification information we kept allowed us to perform this differentiation. In the example outlined above, the mail program and the text editor have different process IDs, so the adjacent records can be separated into their proper place in the different threads representing the mail program and the editor.

As described in the next section, some of our models required gathering data for all invocations of a process, across different machines and sets of users. Doing so required identification of the name of the process, which required clever analysis of the trace, since the system calls we trapped did not necessarily have that information readily available. These models essentially incorporate the file accesses of many different runs of the program by many different users. For example, we gathered data on all invocations of the ls program, a Unix program that lists the contents of directories.

4. Model Building

We gathered the traces described in the previous section to allow us to derive models of user behavior from actual data. Therefore, we started an analysis of the traces early in the project.

The models to be built depended on how we planned to detect insider threats. As with an intrusion detection system, there are two basic methods. First, the system can have a list of known suspicious behaviors. This type of misuse detection requires recognition of a signature behavior among the flood of normal accesses. Second, the system can know about the normal patterns of behavior and look for new patterns out of the ordinary. This type of anomaly detection requires a deep enough understanding of normal behavior to allow recognition of truly abnormal behavior. We built models based on both methods.

A previous intrusion detection system [Lee99] uses an AI approach to predict the next system call based on the previous sequence of system calls for each program. This system uses some ideas that we also use, but it relates to specific exploits and thus cannot be generalized. For example, it detects a buffer overflow exploit in the sendmail program by learning the sequences of system calls invoked by "sendmail". This is not scalable, as it requires recording the sequence for each program in the system. General rules that can be applied to all programs are likely to lead to high false positive/negative rates, since the sequence of system calls is normally specific to each program. We investigated approaches that are likely to scale better and provide more accurate detection.

We began by making some simple models based on listing the files accessed by particular users. This type of model captures a normal working set for each user. By hypothesis, users who make a sudden large-scale change to their working set might be behaving suspiciously.

Our results suggest that the typical user does have a fairly static working set of files. A given user is likely to access some set of files {A,B,C,. . .} with characteristic daily frequency. We have a model for each user that includes the average number of daily accesses to these files.

However, this set is not completely static, and both the number and the percentage of accesses to each file in the set can vary somewhat. Figure 1 to Figure 5 illustrate that the percentage of files accessed per directory for each user does not stay constant over time. In fact, it varies over a large range. (We show accesses to directories, rather than individual files, in these figures because it is easier to view graphically). There are a very large number of files, many of which have very few accesses, and it is hard to represent what is going on for individual files on a chart. The characteristic results are the same for files as for directories.)

Figure 1 shows the number of times files in particular directories were accessed by one user over the course of several days. There are obvious large deviations from day to day, despite all the behavior being proper. Figure 2 shows the same data plotted as a percentage of total accesses. As Figure 3 shows, even a simple count of number of files accessed per day varies widely. The same results apply to other users. Figure 4 and Figure 5 show the percentage and total number of file accesses for a second user. These vary even more. Other results for other users were similar.

One reason for this variation is that the file usage depends heavily on which processes were executed. For example, when a user runs some file "A" a bit more, the file usage will change dramatically if the process opens many files. An example of such behavior is when the process needs to open different configuration files, or the same configuration files repeatedly. Thus, if we build a system based on the distribution of files accessed, the false positive/negative rate will be high.

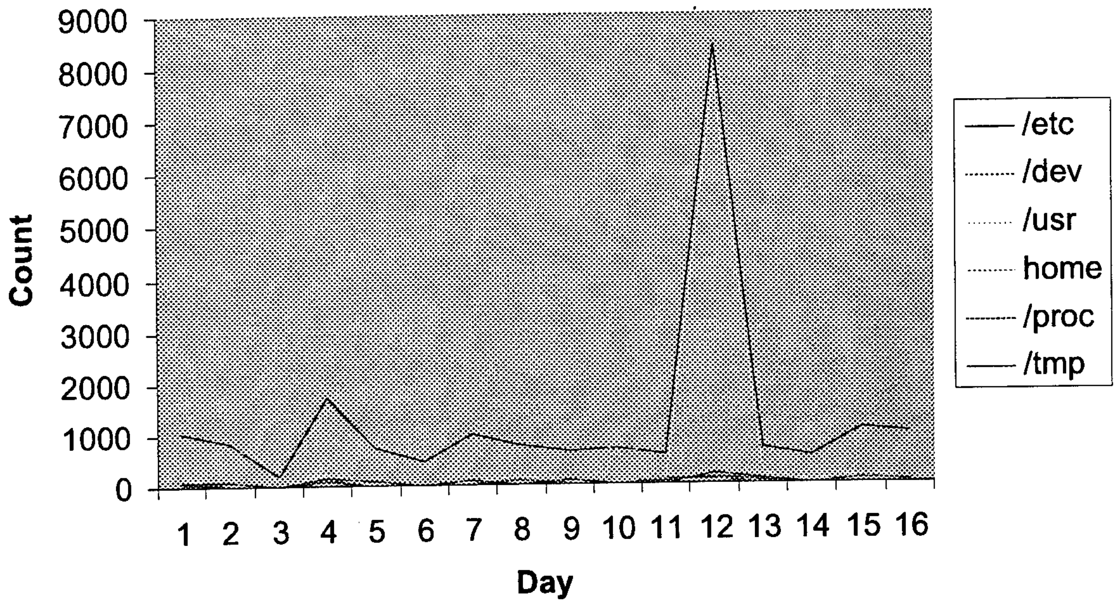

Figure 1. Directories accessed by one user over several days.

Another reason that this model is ineffective is that the number of files opened per user per day is on the order of tens of thousands. (One user, during one day of the trace, opened nearly one million files.) Thus, an exploit that needs to only open a few files to succeed can fool the system, since it will not have any noticeable effect on the distribution of file accesses.

Models based on accesses to individual files are somewhat more indicative if we apply a little more statistical analysis to the data. By including the standard deviation of the number of daily accesses to each file in the model, we can more accurately signal real changes in behavior. Thus, when a day's accesses to a particular file fall more than 50% outside the model's standard deviation of the number of times each file was accessed, it signals a real change in user behavior.

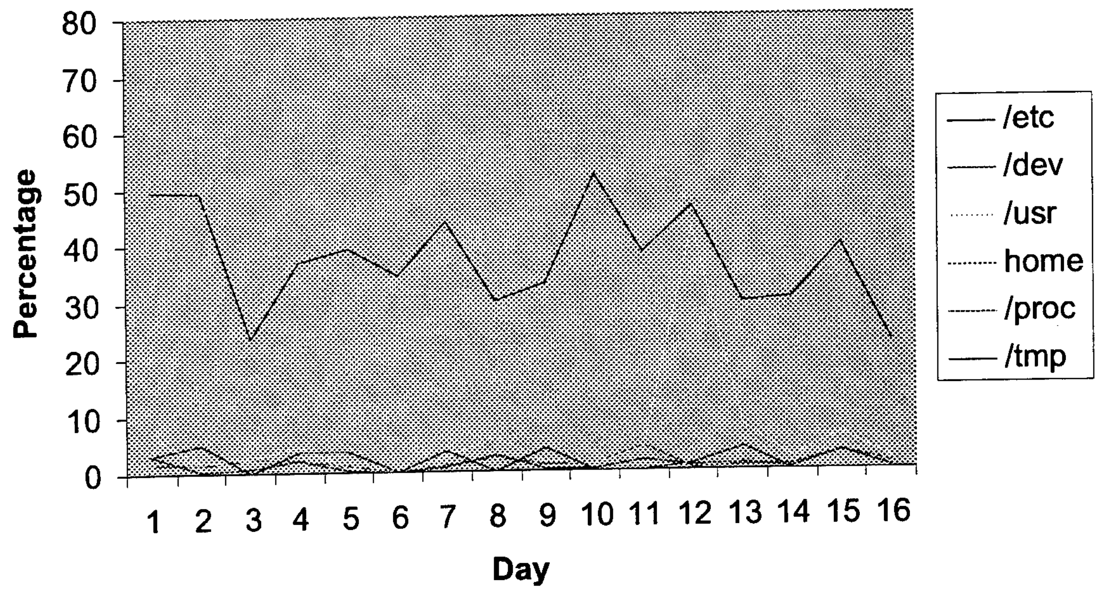

Figure 2. Percentage of files accessed in various directories by one user per day.

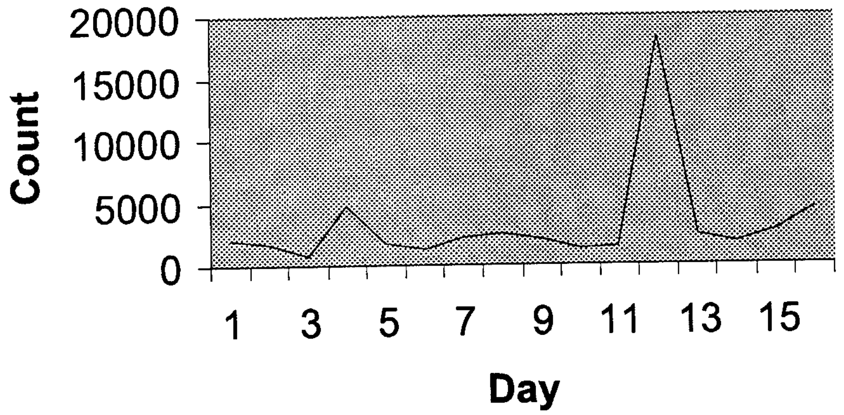

Figure 3. Total number of files accessed by a user over several days.

An analysis of our traces suggests that in most normal cases, when a user's working set behavior varies far outside the standard deviation of normal behavior, it is because the user is working in a new area of his own home directory. Typically, a user's home directory contains files created by the user and completely under his control, so changes in behavior in subdirectories of the home directory are generally not suspicious. Nothing the user does with these files can compromise the security of the machine. This observation shows that applying pure statistical measures to the problem of detecting insider threats is insufficient. Adding a little knowledge about what constitutes safe and dangerous behavior improves the accuracy of the system.

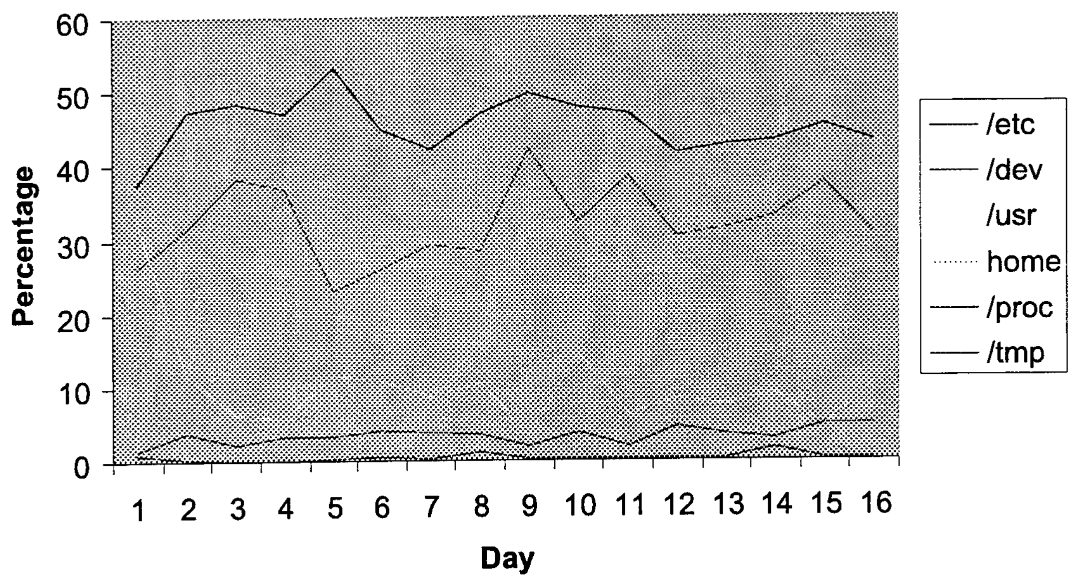

Figure 4. Percentage of directories accessed by User B over time.

As alluded to in Section 2, we traced the behavior of some special daemon users. On a typical Linux system, some special user IDs are created to handle system activities that benefit all users working on the machine. These special IDs include nobody, xfs, and several others. Our analysis showed that such non-human users have very rigid working set behavior patterns. For example, the nobody user typically has 23.53% percent of his open system calls for the file /etc/hosts, 11.76% for /etc/ld.so.cache, 5.88% for /proc/net/tcp, and so on. These special users typically have important system privileges, and thus are often a target of attackers. However, since their behavior can be so rigidly characterized, we can easily detect when attackers have gained access to these privileged accounts. Even slight deviations from their normal patterns will be recognizable. Figure 6 and Figure 7 show the file access patterns for the "bin" and "xfs" users. Note the high predictability of which files will be accessed and in what proportions.

Early in our trace analysis work, we realized that keeping information about process trees would be vital for proper detection of danger. In many cases, the file access profile of a program is even more telling than the file access profile of a user. When an insider is seeking to compromise a privileged program, he is likely to force it to behave in uncharacteristic ways, which may be reflected in its file access profile. If the program is compromised, chances are good that the attacker will use his newfound privileges to exercise the program differently than is normal.

Our analysis shows that 92% of the processes traced have a fixed list of files that they access. In some cases, the percentage of accesses by the process to different files on this list varies, but rarely by more than 20%. For this large class of processes, an attacker will have relatively little room to operate after he has compromised them, since a detection system will rapidly recognize that files are being accessed outside of the fixed list, or files on the list are being accessed in improper proportions.

This information allows us to detect when a process has forked off a child process when it usually does not do so. Many privileged programs are inherently capable of doing very limited things, but if the attacker can convince them to fork off a general execution shell under their privileged identity, compromising the program can gain the attacker general privileged access.

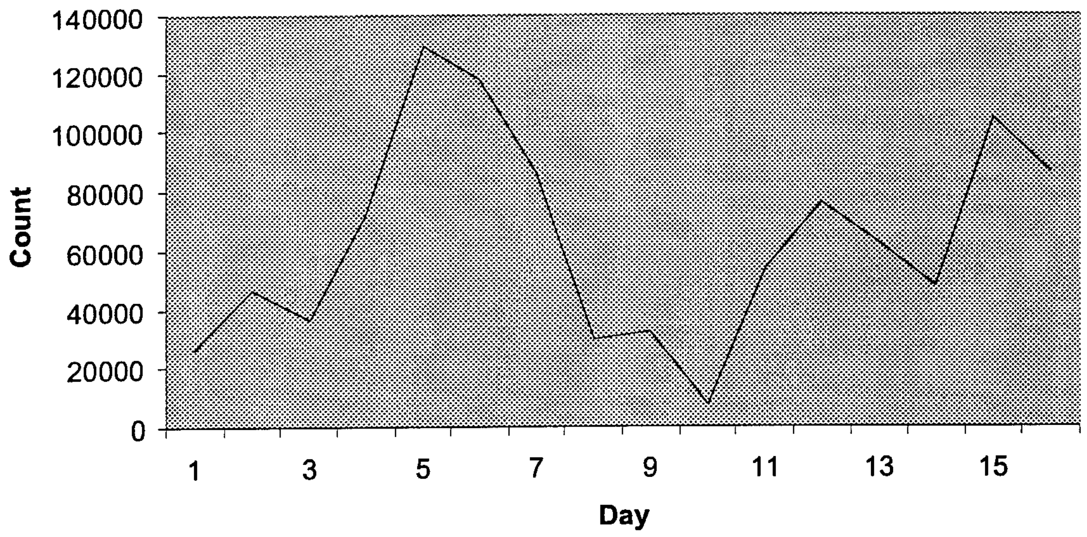

Figure 5. Total number of files accessed by User B over time.

On the other hand, a model based on observing the percentage of file accesses per process was not successful. Just as with the per-user approach, the percentage varies too much to be useful. Many processes open a large number of files, obscuring which variations are significant. To determine which variations are significant would require the system to understand the semantics and use of the files accessed

The list of possible child processes for a given process is nearly as predictable and stable as the process's list of files accessed. Our analysis of traces showed that more than 90% of all programs run in the traced environment have a fixed list of possible child processes. (See Figure 8.) In other words, most programs will only create a limited and highly predictable set of child processes. For programs in these classes, observing them create a child who is not in their normal set is a sign of very suspicious behavior indeed.

Process creation behavior allowed us to develop other models that are useful in detecting suspicious user behavior. For example, just as users have characteristic working sets of files used, they have favorite programs. Our traces show that the frequency with which particular users worked with these programs was quite stable. If a user decides to start misbehaving, in many cases he is likely to use different programs to do so. If he needs extra privileges, he will have to find vulnerabilities in privileged programs, which will generally require him to cast his net beyond the normal set of programs he works with. The most common way of doing so is running attack toolkits that probe a large number of programs, most of which are not used by typical users. Such behavior would be far outside our models of proper behavior. Even if the user is working within his normal privileges, but in a dangerous manner, there is a good chance that the damaging acts he intends to perform will cause him to use programs he normally does not use, or at least use his favored set in different proportions.

The models created for this project were simple and were thus able to be stored and accessed in straightforward ways. For example, the model describing which processes were commonly children of other processes could be represented as a simple set of relations. Each relation is defined as {X, y1, y2, .., yn}, where X is the program name and y_i is an authorized child. For example, a valid relation is `{netscape,netscape}`. This specifies that the program `netscape` can execute another `netscape`, but nothing else. The system reads the input file and stores it in local hash tables. Whenever a process attempts to execute a child, the system compares the child name against its database. If the child is not a legitimate process, the system will raise an alarm.

Other effective models are of similar simplicity in their representation and use. This simplicity is a desirable characteristic since it makes the system faster and uses less memory. Also, a simple system is likely to have fewer bugs than a complex one. We kept these principles in mind when designing and choosing models, and were pleased to find that we could obtain good results with comparatively simple models.

After adding tracing of process creation behavior, we added an analysis of common attacks used by insiders to gain improper privileges. We examined these attacks to discover what their patterns of file access and process behavior would be, so these patterns could be characterized and differentiated from our models of proper behavior. Analyzing common attack tools that exercised buffer overflows and other methods of improperly gaining access suggested further signals to watch for that would indicate possible problems. Generally, our models based on analysis of attack tools are examples of misuse detection, but some are more accurately described as anomaly detection. For example, some attacks require repetitively running programs that have flaws that occur probabilistically, until the flaw happens and the attacker gains control. By counting the number of typical executions of particular programs over time for individual users, we can easily determine that a user is trying this technique.

Another useful activity that we had not originally considered was to examine the length of arguments provided to system calls. Most frequently, arguments provided to system calls are short, even if the interface allows much longer arguments. Buffer overflow attacks, one of the most common methods used to gain extra privileges improperly, usually involve providing extremely long arguments to system calls in the hopes that they will handle long arguments improperly. This technique exemplifies misuse detection, though it also points out that misuse detection techniques can also have false positives. In some cases, a proper system call may require a very long argument, such as a file open that specifies a full pathname from the file system root through seven or eight subdirectories.

One useful enhancement to our research is that of examining and analyzing the data provided to these system calls. For a buffer overflow to cause a security compromise, the attacker must write a system call into the stack that alters the flow of control to a program giving him general access to the machine (such as a shell). Since the system call must be represented as an actual machine language instruction that causes the transfer of control, the argument used in the system call being attacked must contain machine code. Machine code is generally very different in its characteristics than proper arguments to system calls, so one approach to detecting possible buffer overflows is to search the arguments provided to system calls for possible machine code. If machine code is detected, the system call is very likely trying to exploit a weakness in the program being run. More work is necessary to determine exactly how one would use this technique in real systems, but it seems promising.

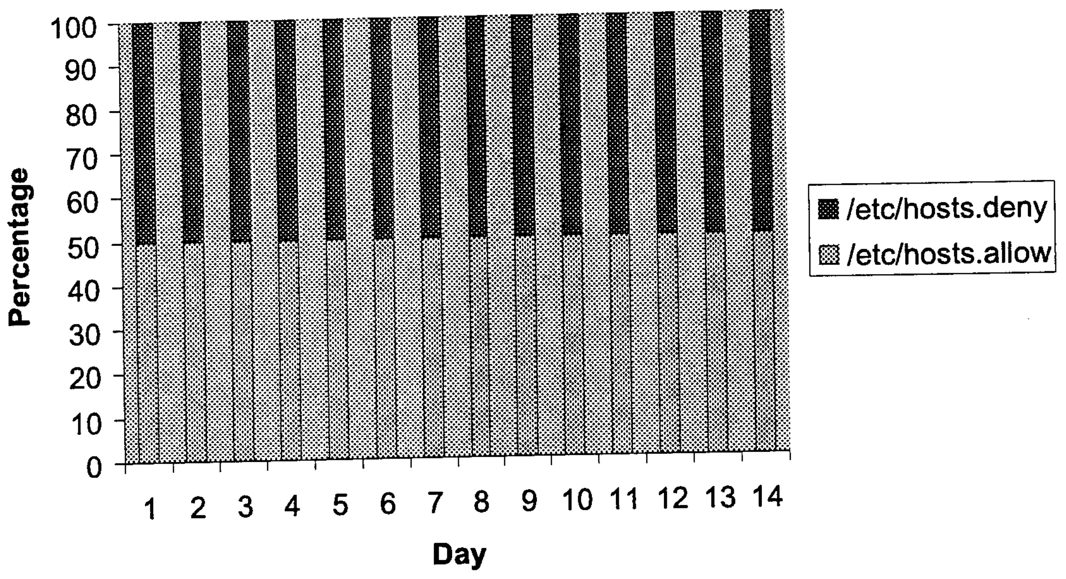

Figure 6. File usage by "bin" user.

Analysis of the reasons for normal behavior also proved helpful. For example, such analysis revealed that sometimes considering the correlation of accesses to different files was useful. Many programs access files in particular patterns, first file A, then file B, then file C, and so forth. If a user runs such a program multiple times, we expect to see correlations between the number of user accesses to files A, B, and C. Speaking practically, if a user typically spends much of his time running compilations, we expect to see correlations between file usages that match compilation correlations. If such a user decides to misbehave, chances are he will start doing something other than compilations. Even if his misbehavior tends to access largely the same set of files, the change in correlations between the number of accesses to various files can signal a shift in user behavior.

This simple description hides many key important details. For example, in many cases user behavior changes gradually over time. Some of the traced users had significantly different profiles after a few months. It's common for users to discover new programs, install upgrades that act differently, or learn to use old programs in new ways. But the frequency with which an average user does any of these things is more likely to be once every few months than several times a day. Figure 9 shows this effect. This figure plots the number of distinct programs run by several traced users over the course of several months. Some users have a practically constant number of programs over the displayed period, while others change dramatically. Note that the graph shows occasional large jumps in the numbers of programs invoked by users over the course of one or two months.

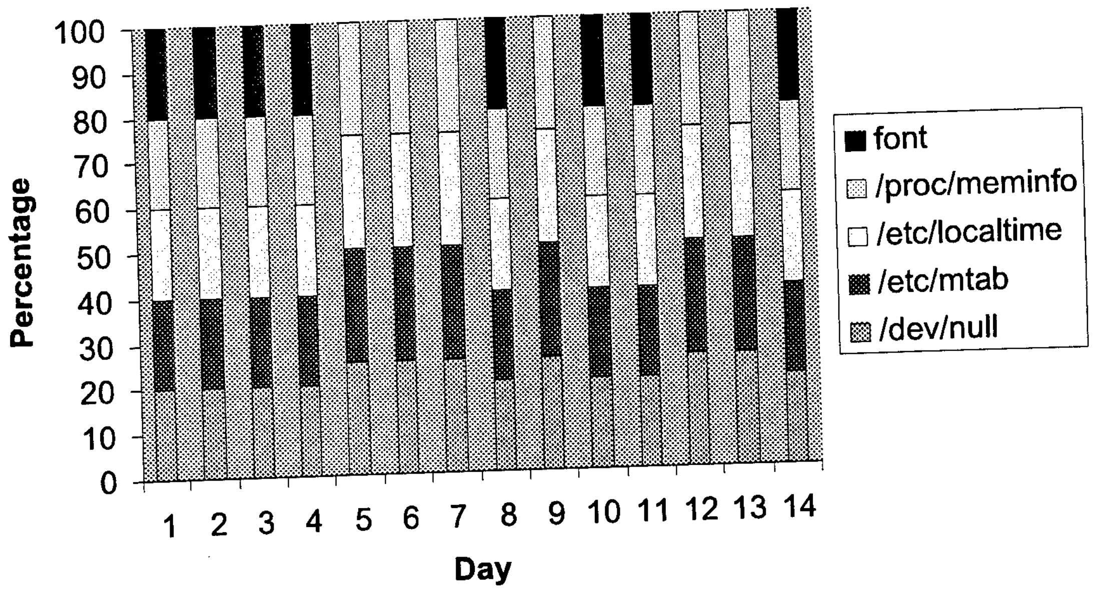

Figure 7. File Usage by "xfs" user.

While changes may be slow, a practical system for using file access activity to detect insider misbehavior would need to alter its models over time, and would need mechanisms to differentiate between benign changes based on new tools vs. malign changes based on attempts to subvert the system.

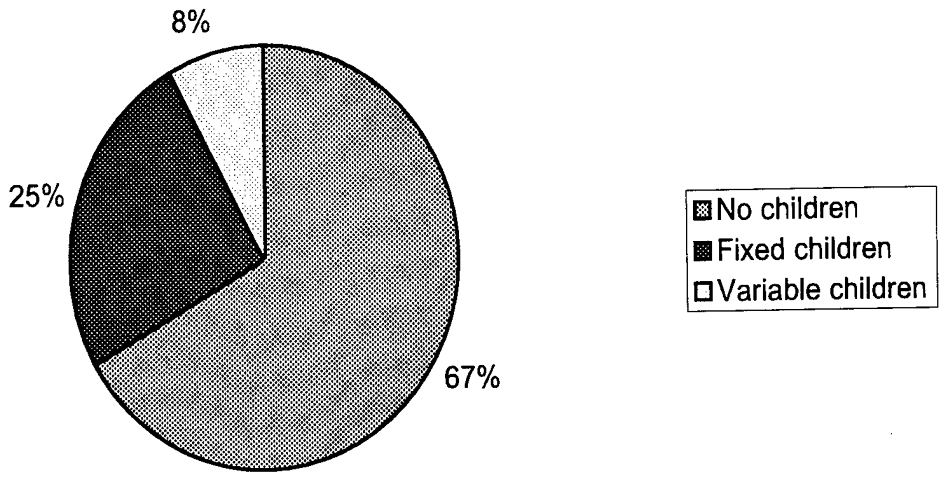

Figure 8. Processes' parental patterns.

Also, user file access behavior is time varying. We see different statistics at 3 a.m. in the morning than at 3 p.m. in the afternoon, and weekend behavior is characteristically different from weekday behavior. A real system would need to take these effects into account.

We did not make use of all information that we gathered in the trace. For example, none of the models we developed made use of the read and write system calls that were traced. Dealing with files at the level of simple access (signaled by an open) is much easier than analyzing the effects of the more detailed read and write system calls, so we started building models without using the read/write calls. As discussed above, we were able to build very effective models with more limited information. We have considered augmenting those models with detail related to reads and writes, and we believe that a full production system might want to observe reads and writes for at least some important cases. However, using this data proved unnecessary for this phase of the research. (Also, as noted later, there are serious performance penalties with tracing and analyzing reads and writes, so a system that can succeed without watching them will be preferable to a system that requires them.)

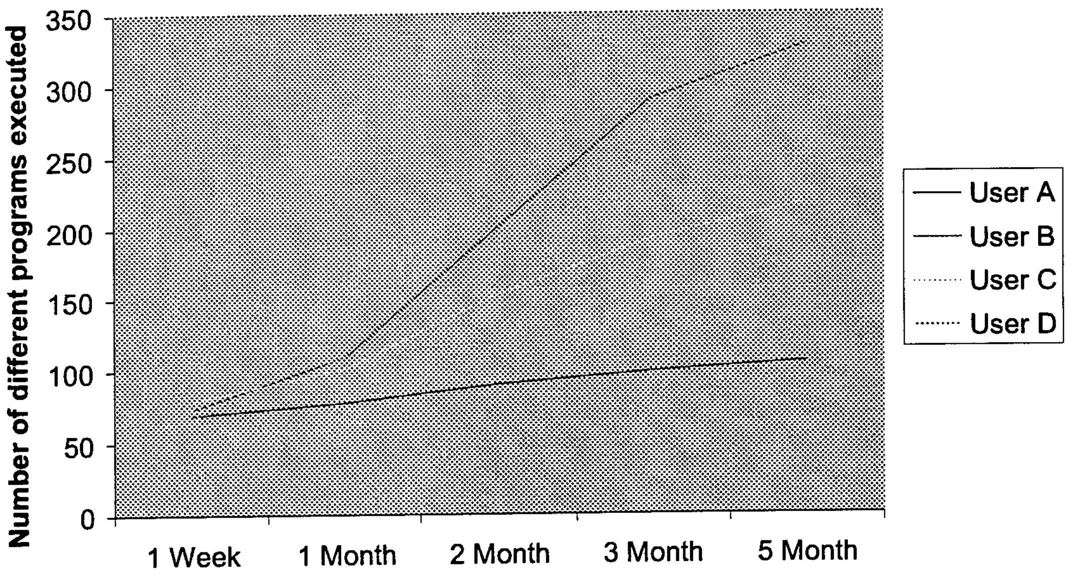

Figure 9. Changes in user behavior over time.

5. *Performance, Related Work and Future Work*

We did not complete all the milestones outlined in the original plan. However, we believe that the work we report here meets the substantial purpose of the project, which was to demonstrate that easily gathered data on file system activity could offer great leverage in detecting insider threats. We will describe here our performance against each milestone in the original project, describe how certain milestones were not completed, and discuss why their purpose was met by other aspects of the research. We will then discuss how this research could be further developed to help defend against insider threats.

5.1 Performance Against Plan

The original proposal contained six milestones, reproduced here:

1. The start of our tracing program. We have existing tools, but expect they will need minor enhancements to gather the information needed for detecting insider threats. We will meet this milestone at the beginning of the second month of the project.

2. The completion of the first tracing cycle. We plan to trace several users for one month, so this milestone will be met at the end of the second month of the project.

3. Development of the code to support several different methods of analysis of the traces and ongoing activity. We will meet this milestone at the end of the fourth month of the project. More such code will be created on an ongoing basis.

4. Development of the simple monitoring system that watches current behavior and compares it to the models derived from the traces. We will complete this milestone at the end of the sixth month of the project.

5. The initial set of results from running the system live, against normal behavior and artificial insider attacks. This milestone will be completed at the end of the eighth month of the project.

6. A final report and demonstration of the complete system, with improvements based on the results of earlier milestones. This milestone will be completed at the end of the twelfth month of the project.

Milestones 1 and 2 were fully met, and actually exceeded. We started tracing before funding was finalized for this project, and continued through the end of the project. Instead of gathering one month's worth of data, we gathered nearly two years worth. As mentioned earlier, this encompassed nearly 1.5 billion file operations, including all file opens, closes, reads, writes, creates, deletes, renames, metadata reads and writes, and other operations, as well as process creation and destruction information. To our knowledge, this is the most extensive file trace of its kind ever taken. After proper sanitization of the trace data, it will be made available to the research community via the web.

This extensive amount of data tracing allowed us to achieve a higher confidence about our results than a single month's trace would have permitted. Also, it allowed us to capture effects that we otherwise would not have seen. As mentioned in Section **Error! Reference source not found.**, real user behavior tends to change over time, but on a fairly large time scale, on the order of longer than one month. A single month's traces would have captured little of this effect.

Milestone 3 was also fully met. This milestone involved development of tools to analyze our traces. While we had built trace analysis tools before, we expected to need new tools to fit our purposes here, and indeed we did. We built various tools to allow us to automatically extract the models described in Section **Error! Reference source not found.** from the extensive traces we gathered. For example, determining the set of files typically used by a particular process required us to scan the traces for various instantiations of that process, find all subsequent file activities related to each instantiation, and collate the resulting data. Each model described in section **Error! Reference source not found.** required this effort, as did a number of models that proved ineffective and were hence not discussed in Section **Error! Reference source not found.**. As also expected, we determined part of the way through our tracing activity that we needed to add some features not present at the start of tracing, such as tracing activities by the root user.

Analyzing large traces is computationally expensive. The analysis code needs to build large tables that take up a lot of memory, and must perform I/O to read each record being analyzed. For this reason, the numbers mentioned here cover only the last six to nine months of the trace, rather than the full period. Even with this limited coverage, running an analysis on a single user to get information on his patterns of file access required a full night.

One component of this milestone, whose importance and difficulty we underestimated, was implicit in the list of milestones, but not stated outright. Merely analyzing the traces and extracting data for models was of no value if we did not develop useful models. We had expected that some very simple models would be fruitful, but the actual data in the traces showed us that we needed a broader set of more sophisticated models. We went through many possible models, finally settling on those described in Section **Error! Reference source not found.**. Merely having the models and being able to pull data out of the traces was still not enough, however. We had to analyze that data and understand its implications to determine that we really could use it for our purposes. For example, in Section **Error! Reference source not found.** we indicated that in excess of 92% of all processes traced had a fixed list of files that they accessed. In addition to gathering the data showing us the list of files used by all of the traced processes, we had to analyze the lists. We then had to determine how to tell, within the fixed file lists, if there was substantial variability in which files were accessed from run to run.

Milestone 4 was not performed as originally envisioned. We did some work to create a system that could observe and detect attacks in real time, but we did not take it to the point where it could be reasonably run in the same way that we ran the tracing code. We were able to use the limited system to detect deviations from our model caused by a particular buffer overflow attack and by an attack based on improperly creating symbolic links to sensitive files. The monitoring system did indeed signal a problem here. But instead of extending this system to the full extent implied in this milestone, we performed a more extensive analysis of several real vulnerabilities and attack tools. This analysis gave us a more accurate picture of the kinds of activities we needed to recognize as possible insider threats. Several of the models presented earlier were developed after examining attack code.

Actually getting real attack code to run on our system was more challenging than we had anticipated. While bugs like buffer overflows are rather common, finding a version of a program containing a vulnerability that ran properly on our current Linux kernel was something of a chore. Like many other responsibly designed and maintained systems, when buffer overflow problems and similar security bugs are discovered in Linux, they are quickly patched and the system developers seek to ensure that the faulty code is no longer available. For most purposes, people do not want code with known security flaws. Because our tracing facility requires kernel patches, we were limited to any buggy code we could track down that ran under the particular version of the Linux kernel we were using, or a substantially similar version. We did eventually find a few programs to actually test with, however, including the buffer overflow attack and the symbolic link attack mentioned above. The difficulty in finding large numbers of attacks that we could actually run on our system was one factor that caused us to devote less time to implementing a monitoring system than originally planned.

Since milestone 4 was altered, milestone 5 was also altered. We did run some attacks against the system, but did not install it on production machines for regular use. Instead, we used the time to develop more models, gather more traces, perform more detailed analysis of the traces, and to study more attack tools.

This report represents milestone 6. This milestone also includes a demonstration, but we have not had a suitable opportunity to demonstrate the system to DARPA. If desired, we can do so, but we do not think it is a particularly valuable effort. The goal of the research was to answer this question: can information about file access behavior be used to detect insider threats? We believe that giving a simple demo of one or two attacks being detected by a very minimal system does not really shed much further light on that question.

Performing this research required us to overcome many technical difficulties. The traces we collected contained a great deal of material, much of which proved to be unhelpful in detecting threats. We had to differentiate between the useful and useless information and remove the latter before analysis could proceed. Many of the useless records could be categorized and removed fairly mechanically, once we had determined their lack of utility. Examples include references to library files (too common to mean much of anything), references to temporary files (generally of no security interest and transitory in nature), and references to files in users' home directories (of no security interest). However, it proved impossible to predict and categorize all such files ahead of time. We thus had to do much hand analysis of the traces to find and remove unimportant files.

Users tend to be quite dynamic in their behavior. As a result, much of the information captured in the traces was not easy to incorporate into models. The fact that our users work on laptops that are sometimes connected and sometimes not connected exacerbated this problem. When connected, users interact with servers and other machines, leading to different behaviors than when they are disconnected and must rely on the local resources of their portable computers. We also found it difficult to find patterns of when our users did and did not work, making window-based analysis tricky.

The traces are huge. There are millions of trace records for every day. Even when known useless records are deleted, the processing time required to reduce a trace to a set of useful models is large. It takes six hours on a fairly powerful computer to analyze a three-month trace for a single user. Limited memory causes much swapping to disk, which slows the analysis down. Every time we needed to test a new idea or an alteration to a model, this process typically had to be repeated. In a working system where a particular set of models had been chosen for observation, this cost would be paid once and lower costs paid to incorporate data on an ongoing basis, so this difficulty is not necessarily a deficiency of the technical approach, but it did require a great deal of time during the research.

As noted earlier, for a given version of the kernel and associated programs, only a certain set of exploits were actually available for testing. We could not usefully test with exploits that work for other versions of the kernel or application programs, because such exploits might never succeed in getting to the point where the system could be expected to detect their misbehavior. They would crash the application or exit in other ways before they did anything suspicious. Generally, responsible parties do not make exploit code readily available, so even those exploits applicable to our system were not necessarily available in a usable form. We did not want to spend a lot of our time trying to reproduce or discover exploits, so we were forced to test with a relatively limited set.

While we worked hard to get good traces, sometimes mistakes occurred and traces were corrupted. Analysis of corrupted traces results in invalid models, if it completes at all. Also, a corrupt trace for a particular user in the middle of the tracing period puts a boundary on how long a continuous period of behavior was available for analysis. As our results show, in some cases the system's performance improves when a longer period of observation is available, and thus corrupt traces limited our ability to investigate such issues.

Each of our machines had a slightly different configuration. They all ran they same kernel, but other details differed. For example, a given user's home directory might have a different name on different machines, despite being the same actual directory. These differences sometimes required manual corrections to traces to allow data from different machines to be compared. Generally, we did not foresee such problems, and thus had to discover the problem after it occurred and go back to fix it.

These are far from the only technical difficulties we faced, but they give a sense of the kinds of problems we overcame, beyond the challenges of dealing with the basic research issues.

5.2 Performance of the System

While this research was intended to prove the concept of detecting insider threats by examining file system traces rather than building a working prototype, several aspects of feasibility required some degree of design and testing of a working system. We report some of those results here.

Part of the thesis was that the false positive and false negative rates of a system based on modeling file system activity would be reasonable. We do not claim that the results obtained so far offer definitive evidence on this question, but we have built a working system, run it on normal activity, and tested it against real attacks, so we have some data to report.

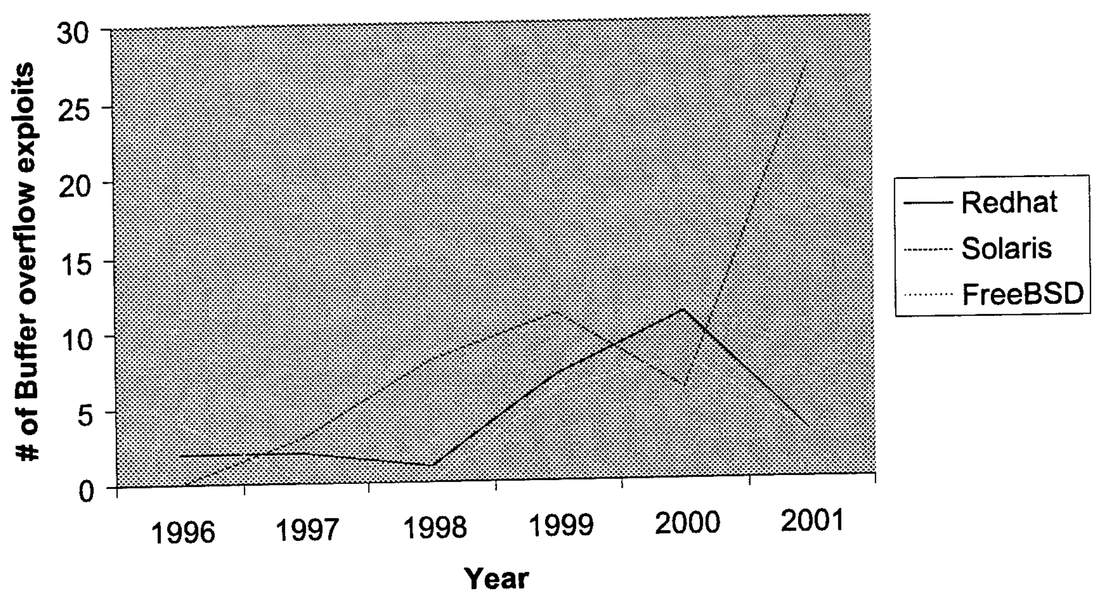

Figure 1. Number of buffer overflow vulnerabilities for several operating systems.

Our tests against live attacks concentrated on buffer overflows, though we also demonstrated our ability to detect attacks of other classes, such as timing attacks using improperly created links. Buffer overflows are commonly recognized as one of the leading causes of security flaws in modern systems [Fordahl01]. In 1998, over two-thirds of all of CERT's vulnerability advisories were due to buffer overflows [Schneier00b]. Figure 10 shows that buffer overflow attacks are common in various operating systems, with a few to a few dozen appearing in each system every year. The trends on this graph do not suggest they are going away. Windows systems are also commonly afflicted with buffer overflows. CERT advisory CA-2002-04, released in late February of 2002, describes a buffer overflow in Internet Explorer under all Windows operating systems [CERT02]. Thus, merely solving the problem of insiders using buffer overflows to obtain extra privileges is valuable.

In our tests, the system has low false positive and false negative rates. The false positive rate is

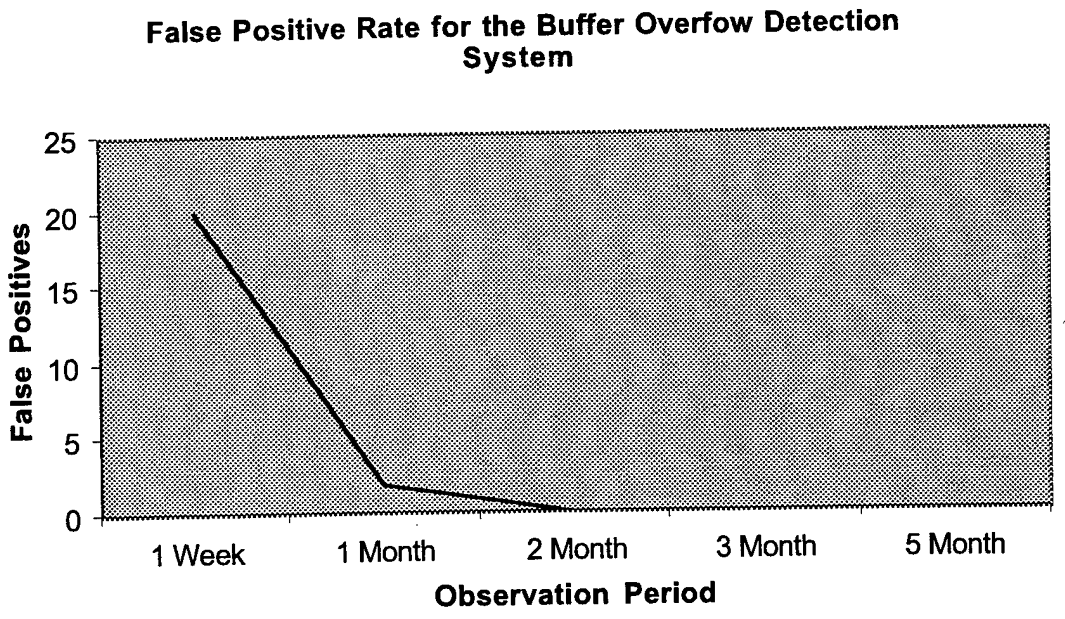

Figure 2. False positive rate for buffer overflow detection system.

dependent on how long the system has observed user behavior and which kinds of exploits it is capable of detecting. As shown in Figure 11, in our environment, with a two-month observation period and a system designed to detect buffer overflow attacks, the system achieved a zero false positive rate. This result is obviously very heartening, but it should be tempered with realism about whether the system can continue to achieve that rate when expanded to handle a broader range of attacks.

Testing for false negatives was somewhat more difficult. A false negative can only occur when an attack occurs and is not detected. Our system deterministically detects (or does not detect) particular attacks. Thus, to determine our system's false negative rate, we needed a collection of different attacks to test it. In practice, each different version of an operating system has a different set of vulnerabilities. Our system worked on a particular version of Linux with particular versions of shared libraries, applications, and associated software. Thus, there were only a limited number of real exploits to test against the system. Also, responsible people do not make attack code readily available, so a working version of an attack was not always available. Spending time trying to find additional exploits ourselves or developing working code to exercise reported attacks seemed like a poor use of research time, so we did what we could with the available set of attacks on one version of Linux. To maximize the number of available attacks, we chose to test this with Linux Redhat 6.0, since we found more exploit code for that version than any other candidate version.

We were able to download and perform seven exploits for Redhat 6 successfully. There are many more exploits for buffer overflows, but they required different versions of libraries and environmental settings. The system detected six of these seven exploits. Bear in mind that the system was not designed to look for specific signatures of any of these exploits – it looked for patterns of behavior that commonly occur when such an exploit is attempted. Many other types of buffer overflows would be detected by this system without any alteration to its code or models.

The model used to detect the buffer overflows uses the observation presented earlier that 90% of all programs have a fixed set of child processes. We were unable to detect an exploit operated on a program not in that category. This exploit attacked `xterm`, a program that must inherently be able to fork and execute a wide and expandable range of other processes, since it provides a basic windowing capability to the user. Other models might succeed in catching this exploit.

Thus, by a simplistic analysis, our false negative rate is 1/7th of all attacks. However due to the small number of exploits being tested, this simplistic analysis does not reflect our true false negative rate. Being extremely conservative, one could estimate the true rate as 10%, since the model in use covered 90% of all programs. However, that analysis presumes that buffer overflow vulnerabilities are equally present in both the set of programs covered and the set not covered. There is no evidence on whether that is true. A scientifically meaningful false negative rate would require a much more exhaustive study of possible attacks.

5.3 Related Work

Much of the related work in this area is in the field of intrusion detection systems. Such systems have a somewhat different goal than our work, but in some cases there are relationships to the methods.

The history of intrusion detection goes back at least to the late 1980s [SSHW88]. Generally, intrusion detection systems are based on vulnerability detection, anomaly detection, or some combination of these approaches. Vulnerability detection is based on looking for known problems. This approach tends to allow accuracy in detecting real problems (since only behaviors that try to exercise known vulnerabilities are flagged), but can only detect already-known attacks. Such systems must be updated periodically to incorporate knowledge of new attacks. Anomaly detection is based on changes in how the observed system behaves. As a result, it can detect previously unknown attacks, but it is more susceptible to misidentifying proper (but new) behavior as an attack [Escamilla98].

RIPPER [Lee99] is a tool that used data mining approaches for automatic and adaptive construction of intrusion detection models. Several types of algorithms were particularly useful for mining audit data:

- *Classification:* Placing data items into one of several predefined categories. The algorithms normally output classifiers, for example, in the form of decision trees. Sufficient amounts of normal and abnormal data need to be gathered to teach a classifier to label or predict new, unseen data as belonging to the normal or abnormal class.
- *Link analysis*: Determines relations between fields in the database records. Correlation of records can serve as the basis for constructing normal usage profiles. An example is the correlation between the command and argument in the shell command history data of users. A programmer may have emacs strongly associated with C files. There are some obvious similarities to our work here, though Lee's approach is more general and does not look as deeply for patterns in file access.
- *Sequence analysis to model sequential patterns.* These algorithms can determine which time-based sequences of audit events occur frequently together. These frequent-event patterns provide guidelines for incorporating temporal statistical measures into intrusion detection models. For example, from the audit data, analyzing network-based denial-of-service attacks requires several per-host and per-services measures.

RIPPER was funded by DARPA, and its creators make strong claims for its performance. However, they concur that in order to detect new or novel intrusions, anomaly detection has to be employed. RIPPER, as a signature detection tool, does not produce signatures of a sufficiently general nature. This project is similar to our project in that it also looked at system calls. A major difference is that it does not take into account the arguments to the system calls, such as file names.

[KRL97] suggests that the intrusion detection system should focus on the correct security behavior of the system, or more particularly on the behavior of a security-privileged application that runs on the system as specified. The authors have designed a prototype that reads the security specifications of acceptable behavior of privileged UNIX programs (e.g., sendmail) and checks the audit trails for violations of these security specifications.

This system also looked at the system call traces and tried to derive a specification for privileged UNIX programs (based on inodes, symlinks, etc.). Their work was not as tightly focused on file system behavior, and concentrated on the proper behavior of individual programs. Our approach looked more generally at all aspects of file access behavior as indicators of proper or improper accesses.

NADIR is another intrusion detection system that uses file system activity as an input [Jackson 92]. Like other intrusion detection systems that consider file system activity, it works off of information stored in normal system logs. This information is far more limited than what we gathered, and thus their analysis of it is equally limited.

There are a vast number of proposed, designed, and built intrusion detection systems [Axelsson 99]. To our knowledge, none have used file system traces to the extent that we have, and none have focused on applying file access analysis techniques to detecting misbehavior by insiders.

5.4 Future Work

This work was intended only as a proof of concept for the idea that insider misbehavior causes noticeable changes in file system activity. Clearly, our results show that it does do so. More work on better models and an examination of further classes of attacks could provide further evidence of this kind, but we feel this concept has been so sufficiently proven at this stage that additional evidence is superfluous.

The obvious next step in this research is taking the very simple system that successfully detected a few attacks into an actual system that detects insider threats using this method, possibly integrated with other methods. To go beyond the already-completed proof-of-concept to this next useful step is a large amount of work, since the added value will be augmenting a real system with a facility that compares ongoing file system accesses to models of proper behavior. Doing so requires a reasonably quick way to determine if the new accesses suggest problems. One result of our existing research is that a single model of file system behavior will not be able to successfully trap all problems, or even all problems in broad classes that are already well understood. Thus, the system would need to compare ongoing activities against more than one model, and possibly against quite a few.

The system could be designed to perform checks in real time or to do analysis of previously logged data at its leisure. The sample system we built checked in real time, but it was only looking at models suitable for detecting the particular attacks tested, and we did no work to determine how expensive it would be to use in normal operations. An obvious question for real-time checks is how much the system would slow down normal access. If the checks are made before the file operation is performed, there would be a delay on every operation checked. The system could test only samples of events, incurring a performance penalty less frequently, but at the cost of possibly missing important warning signals because the events that would bring them to light were not in the selected sample. Depending on the type of models that prove helpful, the system could check against a smaller set of system calls. For example, none of the models we developed used any information about read and write operations. Since reads and writes are more frequent than file opens, which we do observe, the system could only be invoked to analyze open operations and a few other key system calls. Perhaps the costs would be reasonable if only applied occasionally.

Running the analysis after the fact has two problems. First, the signaling of problems would be delayed, possibly allowing the misbehaving insider to perform substantial mischief before the system signaled any problem. Second, the approach of catching up later is only possible if the system has reasonable amounts of idle time at some reasonable point in the future. If not, the analysis will never catch up with the ever-increasing logs of data. If the system is run primarily on single-user workstations, in most situations there would be ample idle time to allow large amounts of analysis. For heavily utilized servers, however, there may never be enough time to analyze logs.

In either case, there is clear value in determining the minimal set of tests required to determine if each file access event represents a threat, while creating a sufficiently low rate of false positives. The traces we have gathered can be used to test any candidate set of models for false positives, and to evaluate the cost of a system during times of normal traffic, but they have the flaw that they include no attacks. One could create a test suite of attacks from known attack code (keeping in mind the difficulties described earlier of getting any particular piece of attack code to work on a given system), but at best, this method would only validate how well the system does against already known attacks. This difficulty in measuring the hypothesized system is by no means confined to this particular research. A common characteristic of security research is the difficulty in measuring its effectiveness.

Another issue for future work is that our traced data represents a single and rather unusual class of users. Most of them were computer science graduate students, and students in systems fields, at that. Such students are vastly more sophisticated and computer literate than the typical user, so there is good reason to believe their behavior is not characteristic of most users. However, we believe that the differences are more likely to work in favor of our method than against it. Sophisticated users are more likely to do new things, install new software, and take shortcuts that may appear suspicious than unsophisticated users. Experience with typical home and office users tends to show that they have a few favored programs and otherwise make little use of the general capabilities of their computers. If that observation holds equally well at the lower level of file system access, these typical users will have more regular behavior than computer science graduate students, so the models that result will be more accurate. While this argument is comforting, it is based on conventional wisdom and informed speculation. Ultimately, it would need to be verified with real data and experience.

One piece of future work that we intend to do is to make our traces available on a web site to allow other researchers to benefit from our work. While making traces available might seem straightforward, the fact that we wish to preserve a reasonable degree of privacy for the users whose activities were traced requires us to perform proper sanitization on the traces before releasing them. Such sanitization would require, at a minimum, concealment of user identities, but should also obscure the file names. However, if one obscures all file names through a randomization process, valuable information is lost with little resulting privacy benefit. For example, there is no particular privacy requirement to conceal the normal sequence of operations when the mail daemon starts downloading messages, but some researchers might find it very helpful to pinpoint these events in the trace. Some care will be required to decide exactly how much to sanitize the traces to allow the best tradeoff between research benefit and user privacy.

6. Conclusions

This research project successfully demonstrated that a user's file access behavior can be analyzed to determine when he stops behaving properly and starts engaging in suspicious activity. The performance costs of gathering the data are acceptable, demonstrated by the fact that over a period of two years we received no complaints related to system slowness. (On previous projects where the experimental system was not sufficiently fast, our local users have never been reluctant to complain about the impact of testing on their work.) The amount of data we gathered was vast, but a real system need not keep all gathered data, and could probably reduce batches of traced records to model data frequently. Further, some of the data we traced has, so far, given us no advantage in detecting insider threats, so a real system would not need to gather this data.

Demonstrating that the method can be used in an experimental environment and actually making it work in a real environment are two different problems. We have only addressed the first. More research and development will be necessary to deal with the second. This project essentially suggests that this additional research and development would be worthwhile.

References

[Axelsson99] Stefan Axelsson. "Research in Intrusion-Detection Systems: A Survey," Technical Report TR: 98-17, Chalmers University of Technology, Sweden, 1999.

[CERT02] CERT Advisory CA-2002-04. "Buffer Overflow in Microsoft Internet Explorer," Computer Emergency Response Team, February 2002.

[Escamilla98] Terry Escamilla. "Intrusion Detection," Wiley Computer Publishing, 1998.

[Jackson92] K.A. Jackson. "NADIR: A Prototype System for Detecting Network and File System Abuse," *Proceedings of the 7th European Conference on Information Systems*, November 1992.

[KRL97] Calvin Ko, M Ruschitzka, and Karl Levitt. "Execution monitoring of security-critical programs in distributed systems: A specification-based approach," *Proceeding of the 1997 IEEE Symposium on Security and Privacy*, Volumn ix, pages 175-187, Oakland, CA, May 1997.

[Lee99] Wenke Lee. "A data mining framework for building intrusion detection models," *IEEE Symposium on Security and Privacy,* pages 120-132, Berkeley, California, May 1999.

[Power02] Richard Power. "2002 CSI/FBI Computer Crime and Security Survey," *Computer Security: Issues and Trends*, Vol. VIII, No. 1, Spring 2002.

[Fordahl01] Matthew Fordahl. "Buffer Overflows Pose Computer Security Threat, Experts Say," AP Online article, December 21, 2001.

[Schneier00a] Bruce Schneier. *Secrets and Lies: Digital Security in a Networked World*, Wiley Computer Publishing, 2000.

[Schneier00b] Bruce Schneier. "Computer Security: Will We Ever Learn?" Cryptogram newsletter, May 15, 2000.

[SSHW88] Michael M. Sebring, Eric Shellhouse, Mary Hanna, and R. Alan Whitehurst. "Expert systems in intrusion detection: A case study," *Proceedings of the Summer USENIX Conference,*page 74-81, Baltimore, Maryland, 17-20 October 1988, NIST.

Appendix A: Samples From File Traces

In the following paragraphs we provide information about the meaning of each field in the trace lines, and give example lines from the file system traces gathered for this project.

Each trace line represents one system call. Most lines represent file-system-related system calls, but we also traced other system calls related to process execution and a few other things. Some of these extra system calls were traced because we needed to know that those particular system calls were invoked; some were traced because they contained information needed to properly analyze the other lines in the trace.

Each line has the same format. The data was stored in ASCII, which allowed us to analyze it more easily by hand. The debugging value alone of this choice made it worthwhile, but it also permitted easier analysis of what a string of related system calls were actually doing. We paid a cost in the amount of memory used and the time required to create a trace line, but the benefit outweighed the cost. Each line in the trace consists of several fields, separated by white space.

Fields of each line

1. *SequenceNo.* A count of how many system calls have been made since the start of the trace. A gap in sequence numbers between adjacent lines in the trace means that there are some system calls not represented in the trace. These unreported calls could be:

- Untraced types of system calls
- System calls invoked by the insider threat detection system itself

2. *UID.* ID of user who executed the system call

3. *PID.* Process ID of the process that executed the system call

4. *Process Name.* The name of the program represented by the process. This information is not readily available in the kernel in all cases, so the system merely does its best to provide the string. Where the name is not available, this field is filled with "??". The most common such case is a process that was forked before the observation component of the insider threat detection system was started during system boot.

5. *Order flag.* An "A" in this field means that the entry is recorded after the system call is executed. A "B" means that it is recorded before the system call is executed. Generally, we record the system call after it has executed, because (as you see below) we wish to include the return value from the system call. Some system calls, however, do not return. For such system calls, we record their entry before they execute.

6. *Timestamp.* A timestamp in seconds and milliseconds

7. *String* describing the system call and its arguments

8. *Return value* of system call

9. *Repetitions.* Sometimes the same system call is repeated several times with no intervening system calls. This optional field keeps a count of how often the system call was repeated, with only exact repetitions counted. If the field is absent, the system call was performed once. If the

field is present, it was performed the number of times indicated. There will not be separate lines representing each repetition, in this case, just the count. The most common use of this field is to compress many repetitions of read system calls, since it is common to repetitively read a file one block at a time until the entire file has been processed. This optimization greatly reduces the size of the traces.

One example line

104 UID 0 PID 812 /sbin/syslogd A 1008288727.437576 open("/etc/ld.so.preload", O_RDONLY, 0) = -1 (2)

This was the 104th system call since the trace started. The UID was 0 (the root user). The PID was 812. The process that performed the system call was /etc/syslogd. The next field, containing "A," indicates that the record was created after the system call returned. The long number is a timestamp. The system call was open ("/etc/ld.so.preload", O_RDONLY,0), meaning it was a call to open a file for read. Since it returned –1, the system call failed. The "2" in parenthesis at the end of the line indicates that the system call was made twice in a row in exactly that form, with exactly those results. Presumably, upon failure to open the file, syslogd made some adjustments and tried again, only to fail again. At this point, either it gave up or made further adjustments that allowed it to succeed. (In which case, the next line in the trace would be identical, except that the return value would be 0, not -1.)

Traces representing normal activity

The lines below are extracted from a trace of normal activity. In particular, it is activity occurring immediately after the machine is booted and a fresh trace file is started:

```
0 RESTART (scratch) at 1008288727 Thu Dec 13 19:12:07 2001
32 UID 0 PID 812 ?? B 1008288727.376532 execve("/sbin/syslogd") = 0
76 UID 0 PID 812 /sbin/syslogd A 1008288727.437433 execve("") = 0
104 UID 0 PID 812 /sbin/syslogd A 1008288727.437576
       open("/etc/ld.so.preload", O_RDONLY, 0) = -1 (2)
160 UID 0 PID 812 /sbin/syslogd A 1008288727.437633
       open("/etc/ld.so.cache", O_RDONLY, 78748) = 3
216 UID 0 PID 812 /sbin/syslogd B 1008288727.437650 close(3, 78748) = 0
252 UID 0 PID 812 /sbin/syslogd A 1008288727.437706
       open("/lib/i686/libc.so.6", O_RDONLY, 5772268) = 3
308 UID 0 PID 812 /sbin/syslogd A 1008288727.437723 read(3, 1024) = 1024
344 UID 0 PID 812 /sbin/syslogd B 1008288727.437844 close(3, 5772268) = 0
380 UID 0 PID 812 /sbin/syslogd A 1008288727.439369 chdir("/") = 0
412 UID 0 PID 812 /sbin/syslogd A 1008288727.439558
       open("/var/run/syslogd.pid", O_RDONLY, 0) = -1 (2)
472 UID 0 PID 813 /sbin/syslogd A 1008288727.439762 fork(812) = 0
504 UID 0 PID 813 /sbin/syslogd A 1008288727.442123
       open("/var/run/syslogd.pid", O_RDONLY, 0) = -1 (2)
564 UID 0 PID 813 /sbin/syslogd A 1008288727.442205
       open("/var/run/syslogd.pid", O_CREAT|O_RDWR, 0) = 0
624 UID 0 PID 813 /sbin/syslogd A 1008288727.442593 write(0, 4) = 4
660 UID 0 PID 813 /sbin/syslogd B 1008288727.442605 close(0, 4) = 0
696 UID 0 PID 813 /sbin/syslogd A 1008288727.442911
       open("/etc/resolv.conf", O_RDONLY, 94) = 0
752 UID 0 PID 813 /sbin/syslogd A 1008288727.443008 read(0, 4096) = 94
788 UID 0 PID 813 /sbin/syslogd A 1008288727.443057 read(0, 4096) = 0
824 UID 0 PID 813 /sbin/syslogd B 1008288727.443077 close(0, 94) = 0
860 UID 0 PID 813 /sbin/syslogd A 1008288727.443280
       open("/etc/nsswitch.conf", O_RDONLY, 1750) = 0
```

```
916 UID 0 PID 813 /sbin/syslogd A 1008288727.443333 read(0, 4096) = 1750
952 UID 0 PID 813 /sbin/syslogd A 1008288727.443468 read(0, 4096) = 0
```

This trace fragment shows several files being open, read, and written. The files in question are opened by a program called `syslogd`, which is the next program executed in the Linux boot sequence on our installation after the insider threat detection system has been started. Note (in line 916) that the `read()` system call was repeated 1750 times to read all of `/etc/nsswitch.conf`.

Traces representing attack traffic

The lines below were captured during an attempt to execute an exploit that attacks a weakness in the `cron_root` program. This program is used to run tasks periodically in the background, allowing the system or users to schedule tasks for execution in the future.

```
785516 UID 0 PID 546 /tmp/cron_root A 1015623540.370271 open("/etc/passwd",
      O_APPEND|O_CREAT|O_WRONLY, 744) = 4
785564 UID 0 PID 546 /tmp/cron_root A 1015623540.370459 dup2(4, 1) = 1
785600 UID 0 PID 546 /tmp/cron_root B 1015623540.370501 close(4, 744) = 0
785636 UID 0 PID 546 /tmp/cron_root A 1015623540.370659 write(1, 44) = 44
```

Here, the attacker has previously used a buffer overflow to gain control of `cron_root`, and is now trying to use the root privileges he has obtained to open the password file for writing. This is an attack we would catch, because we know that the cron_root program never opens this file for writing. Our model would show that the program was using a file it normally never uses.

In the following trace fragment, we see the attacker trying to cover his traces by removing a number of temporary files. The first line shows the `execve` system call being used to start the `rm` program, which is used to remove files. The `lstat`, `access`, and `unlink` calls are the normal steps taken by `rm` to remove a file. They are invoked repetitively on `/tmp/cron_echo`, `/tmp/ce`, and `/tmp/cron_root`, removing those three files.

```
789296 UID 0 PID 547 /tmp/cron_root B 1015623540.392235 execve("/bin/rm") = 0
789632 UID 0 PID 547 /bin/rm A 1015623540.395261 lstat("/tmp/cron_echo") = 0
789676 UID 0 PID 547 /bin/rm A 1015623540.395367 access("/tmp/cron_echo") = 0
789720 UID 0 PID 547 /bin/rm A 1015623540.395504 unlink("/tmp/cron_echo") = 0
789764 UID 0 PID 547 /bin/rm A 1015623540.395555 lstat("/tmp/ce") = 0
789800 UID 0 PID 547 /bin/rm A 1015623540.395594 access("/tmp/ce") = 0
789836 UID 0 PID 547 /bin/rm A 1015623540.395639 unlink("/tmp/ce") = 0
789872 UID 0 PID 547 /bin/rm A 1015623540.395681 lstat("/tmp/cron_root") = 0
789916 UID 0 PID 547 /bin/rm A 1015623540.395719 access("/tmp/cron_root") = 0
789960 UID 0 PID 547 /bin/rm A 1015623540.395765 unlink("/tmp/cron_root") = 0
```

APPENDIX B

PRIOR MONTHLY REPORTS

File Profiling for Insider Threats
Contract F33615-00-C-1746
November 2000
Contractor's Progress, Status and Management Report
Line Item 0001, Data Item A003
Reporting period: 6 November to 30 November 2000

This report covers the first month of performance under this contract. Due to issues concerning administrative details of the contract, we did not begin actual work on the contract until fairly late in the month. So far, our activities have been largely confined to defining more concrete plans concerning how this research will be conducted.

Our basic method requires that we digest file traces gathered for other purposes to build models of appropriate normal insider behavior. We have a large amount of recently gathered trace data of this kind and have produced software that preprocesses the trace to make it easier for serious analysis. For example, this software canonicalizes file names in the trace, ensuring that all occurrences of the same file in the trace are properly recognized. Since the trace records sometimes list file names as full path names from the file system root and sometimes as shorter names relative to the current working directory, this canonicalization is not trivial. The software also performs other simple cleanup tasks on the traces.

We will shortly begin work on more carefully defining the models that we will develop from the trace. This work will involve moving from the sort of model concepts discussed in the proposal to detailed examinations of what information we will need to extract from file trace records and what sort of processing will need to be performed to produce the model from that information. This work is a necessary precursor to actually building the models.

We are also recruiting a graduate student to perform the bulk of the research. The success of the research will depend in large part on the quality and interest of the student assigned to do the research, so we are recruiting very carefully. We anticipate that we will select a graduate student from the incoming class of the winter quarter, so our choice might not be completed until January. We believe that the startup time for a new student on this project should be fairly short, as useful work can begin without the student absorbing too many details of the operations of existing software. Thus, we expect to begin development of software embodying models of proper behavior in January. In the meantime, we will continue to work with the existing traces to ensure that they will contain the information we need in the form required.

To date, we have not spent any contract dollars. We expect to begin charging to the contract in December.

FILE PROFILING FOR INSIDER THREATS

UCLA

December 2000

BAA-00-06-SNK; FRT Research Topic 2

Contract F33615-00-C-1746 (LI 0001, Data Item A003)

1 December – 31 December 2001

Technical Point of Contact:
Peter Reiher
reiher@cs.ucla.edu

1. Introduction and Overview

This project is investigating the possibility of detecting the misbehavior of authorized users by analyzing patterns of file system access. Our thesis is that normal, well-behaved access is characteristically different than the forms of access used by an insider trying to step beyond his legitimate role. To do this, we must trace large quantities of legitimate access, derive models of appropriate behavior from those traces, and compare these models to sample misbehaviors.

The project is expected to 15 months, overall. We were fully funded at $94,154. Major team members are Dr. Peter Reiher, Dr. Geoff Kuenning, and graduate student Nam Nguyen.

2. Project Research Objectives

We must determine if normal access can be easily modeled across many users and across time. We must also determine if improper behavior actually does produce distinguishably different file system behavior.

3. Expected Project Results

If successful, our project will demonstrate that it is possible to detect important forms of insider misbehavior by intelligently observing file system behavior.

4. Achievements and Issues

Our progress was relatively slow this month, due to the holidays. Our primary effort for the month was dedicated to investigating characteristic patterns of one particular kind of attack, a buffer overflow. We looked for patterns in the behavior of the fork() and exec() system calls, since the most common behavior of an attacker after succeeding in overflowing the buffer of a privileged program is to fork a shell from that program. Doing so allows the attacker to execute arbitrary commands with the privilege of the attacked program.

In some cases, the attacked program legitimately issues fork() and exec() system calls. However, we believe that the legitimate uses of these system calls can be distinguished from attack uses by their position in the program's file access traces. In addition to keeping track of which file system calls have been executed, these traces are essentially sets of markers of characteristic paths of execution through a program. Other system calls, like fork() and exec(), may legitimately occur at particular places in these paths, but are suspicious when they occur at other places in these paths.

We also investigated a tool that purports to perform automatic classification of trace data. Such a tool would be useful for our purposes, since it would output distributions of the occurrences of particular types of system calls and other patterns of behavior. This particular tool proved inadequate. It requires huge amounts of data to properly train the program to recognize patterns for classification, and it requires a particular table format to be created manually before it can do any training. We are unlikely to use this tool.

Progress on statement of work milestones:

1. Tracing program. We have gathered large quantities of file traces. Tracing continues on several machines in our office.
2. Completion of first tracing cycle. Completed. Models have been developed.
3. Development of code to support data analysis. All code that we know is needed is completed. We expect periodically to find a need for more such code.
4. Development of simple monitoring system. This activity is nearing completion.
5. Initial results from running the system live. Work continues on this milestone.
6. Final report. This final report will be written when all other tasks have been completed.

5. Expenditures

December expenditures	Total: $12,186 Salary + related benefits and OH: $11,938 Networking & supplies: $248.00
Total expended to date:	$76,725 (81.5% of funds)

Remaining funds are adequate to complete all work by contract end (5 Feb 02)

6. Program Status

Contract completion date: 5 February 2002
Percentage of contract period that has elapsed: 93%
Total funds that have been expended: $76,725
Percentage of contract funds that have been expended: 81.5%

File Profiling for Insider Threats
Contract F33615-00-C-1746
March 2001
Contractor's Progress, Status and Management Report
Line Item 0001, Data Item A003
Reporting period: 1 March to 31 March 2001

We have made good progress on our project along the lines indicated in the previous month's report. We are investigating effective file access models for characterizing proper behavior. We are also improving the tools we use to gather and analyze our trace data.

Investigations continue on the models that describe the proper set of files for particular processes to access and the proper set of processes to access particular files. The first model analyzes all files that are accessed by a particular type of process (such as an editor, a web browser, or a compiler). Since there are multiple invocations of many of these programs in a single day's trace, and even more over the course of multiple days, we must eventually merge the results from individual runs. Attackers frequently seek to gain privileges or access protected data by intentionally misusing programs to make them behave improperly. We hope that the list of files used during normal behavior will be very different than those used during such attacks.

For each of these programs, there is likely to be one set of files that is inherently characteristic of running that program for any purpose, and a second set closely related to the particular invocation. For example, an editor opens a number of libraries and configuration files regardless of which file is being edited, but editing your resume obviously accesses some different files than those accessed when editing your monthly report. The challenge for this model will be to filter out these properly different files from the files that are accessed when the attacker is attempting to use the program to compromise the system.

The second model looks at all programs that ever access a particular file. Merging the results from multiple runs of programs is far easier for this model, since the model does not need to differentiate the behavior of different runs of a program. This model essentially requires us to keep a record of all files touched by any run of a particular program. If the model is produced during periods of good behavior, new additions to the list might be the result of trying to misuse programs to obtain improper access to files. For example, if the editor never normally accesses the password file (which is typical, since it's preferable to use special tools to update the password file), suddenly having it do so might be a signal that an attacker is at work.

We expect that this model will eventually be augmented with probabilities of access for each program, or counts of how often a particular program accesses a file during a given run, or perhaps more sophisticated models of proper program access to the file. In many cases, programs access sensitive files for perfectly legitimate purposes. However, they can also try to access the same files for illegitimate purposes. These more detailed models may catch such behavior. For instance, perhaps the web browser always accesses a user's security preferences once for read at startup time, to ensure that they are enforced during the web browsing session. However, an access for write later in the middle of the session may be a sign of trouble.

We will also be developing other models, as time and ingenuity permit. Until we have a promising result, we will continue to work with a single user's trace, as discussed in the last report. When we have something that looks good for the single user, we will test it against traces of other users to see if it holds up in general.

Our work with the traces has shown the weakness of some of our tools for dealing with them. The traces are very large. One trace for a user for a single day can take up to 200 Mbytes of storage in ASCII format. Obviously, we need automated tools for handling this. Our initial tools produced the traces in a zip format, but converted them to ASCII format for use. This format had the advantage of simplicity, and it allowed our new student to work with plain text data until he understood how things worked. However, because the files were so large, the actual data analysis was always performed by a program, not by hand. So we were converting from a zip format to ASCII to an internal binary format, which was performance expensive.

Now that our student understands the system well, he has altered his analysis tools to extract data directly from the zip format, bypassing the ASCII conversions. This change allows his tools to run 100 times faster, and allows us to store smaller trace files. Also, handling certain special cases is easier in this format.

Another recent change is that we are now figuring out the process name, rather than identifying the process by PID. Internally, operating systems use a process identification number (PID) to keep track of processes. The process structure has the name of the process available, but dealing with a number is more convenient for the system. When file system calls are made, the process identifier attached to them is the PID. By the time that we analyze the traces, the internal operating system structures that allowed mapping of PIDs to actual names is long gone. Thus, determining that a process was actually Netscape, rather than PID 217, is not trivial.

But it is possible. To permit mapping of PIDs to process names, in addition to file system access calls, we have trapped Linux forks and execs. These are the system calls that create new processes. They too work with PIDs, not names. However, there is a canonical pattern of file system access related to forks and execs. To execute a new program, the file containing the executable must be opened shortly after a new process has been forked. Proper analysis of the traces allows us to deduce the name of each new PID. We have recently added code to the system that allows our models to work with these names, rather than the previous PIDs. Since a given program is assigned an essentially random PID when it starts execution, this step was necessary to merge traces for multiple program runs, as well as to determine actual identities of processes.

Our research for April will continue along these lines. We will refine and improve our models, possibly developing new ones. We will continue to collect traces on multiple machines. (For the moment, most of these traces are being archived for future use.) We will improve our tools.

March Accomplishments

1. Improved two models for file system analysis.
2. Improved the speed and efficiency of our central analysis tool.
3. Started analysis using process name, rather than PID.

April Planned Work

1. Further improvement of existing models.
2. Development of new models.

Financial Data

1. Funds expended through March: $20,200
2. Percentage of total funds expended: 21.5%
3. Number of months remaining on contract: 10
4. Spending plan: On Track

File Profiling for Insider Threats
Contract F33615-00-C-1746
April 2001
Contractor's Progress, Status and Management Report
Line Item 0001, Data Item A003
Reporting period: 1 April to 30 April 2001

We continue to investigate models for detecting insider misbehavior by analyzing file traces. We have worked on several new models, and have tried checking some of our models on more than one user to determine if they are capturing per-user behavior or more generally characteristic behavior.

We have developed one model that should allow us to detect some realistic attempts to obtain privileges that a user should not have. Often, on Unix-style systems, the first step for a misbehaving insider is to gain more file access privileges than he should legitimately have. This is typically done by attacking a program that has extra privileges. Such programs are intended to give unprivileged users stylized access to shared, protected resources. Common examples include print daemons and mail handling programs. Attackers frequently try to force such a program to malfunction in a way that leaves them with the general, unrestricted privileges of that program.

A common example of this is causing a buffer overflow in such a program that results in the unprivileged user being given a command shell under the program's identity. The user then has the ability to do whatever the program's full privileges allow. Often, in Unix systems, this amounts to being able to do anything at all, since many such programs run under the superuser identity. Another common attack is to cause odd timing of program interactions to allow the user to obtain a command shell.

One characteristic of both attacks is that they require the user to interact with the program in unusual ways. We have developed a model that captures the typical file access behavior of many programs, and our analysis shows that they frequently have highly stylized access behaviors. We believe that this model can detect many circumstances where misbehaving users attempt buffer overflows or timing attacks on programs.

In the upcoming month, we will examine some real attacks of this kind and test them to see if our hypothesis is correct. We will also continue to develop other models.

April Accomplishments

1. Developed model that has promise for detecting certain common types of attacks

2. Examined several models across different users
3. Began investigating candidate attacks for testing models

May Planned Work

1. Test promising model against suitable attacks
2. Continue developing new models

Financial Data

1. Funds expended through April: $24,410
2. Percentage of total funds expended: 25.9%
3. Number of months remaining on contract: 9
4. Spending plan: On Track

File Profiling for Insider Threats
Contract F33615-00-C-1746

May 2001

Contractor's Progress, Status and Management Report
Line Item 0001, Data Item A003
Reporting period: 1 May to 31 May 2001

This project involves building models of user behavior to detect the difference between proper and improper insider behavior. These models are based on file system traces taken over the course of several months in our labs.

As reported last month, we have found a model that has promise for detecting certain common kinds of attacks, such as many buffer overflows or timing attacks on temporary file creation. In essence, this model encodes the typical file access behavior of a program. Our hypothesis is that buffer overflow attacks and timing attacks try to use programs in unusual ways that would not match the model of normal behavior.

We spent this month refining the model and preparing a test to see if our hypothesis is correct. The preparations required obtaining a sample attack to test against the model. While sample attacks are readily available on the network, there are some practical issues that make it slightly difficult to find an appropriate one, such as finding one that runs on Linux and on an appropriate version of the Linux kernel. Our system relies on a kernel hook that we have only installed on a couple of versions of Linux, and going back in time to resurrect an old kernel and add the hook is time consuming.

We hope to have a suitable sample attack installed on our system and tested in the near future.

We may have a slowdown of progress over this summer, as our key student researcher will be out of the country for most of the summer. He will be taking a computer with him to work while away, but he will have limited connectivity.

May Accomplishments

1. Improved candidate model to start testing for detection of misbehavior
2. Investigated several candidate attacks for testing.

June Planned Work

1. Choose one or more attacks for testing.
2. Perform preliminary testing.

Financial Data

1. Funds expended through May: $27.657
2. Percentage of total funds expended: 29.4%
3. Number of months remaining on contract: 8
3. Spending plan: On Track

File Profiling for Insider Threats
Contract F33615-00-C-1746
June 2001
Contractor's Progress, Status and Management Report
Line Item 0001, Data Item A003
Reporting period: 1 June to 30 June 2001

This project involves building models of user behavior to detect the difference between proper and improper insider behavior. These models are based on file system traces taken over the course of several months in our labs.

Because of final exams and the key student leaving the country for the summer, progress during the last month was slight. We continue along the same lines as reported in the last month's report. We have a promising model using our basic methods that we believe can detect certain common classes of attacks by examining file system activity.

We continue to look for a suitable test attack to check our hypothesis. The major difficulties are practical - finding an attack of the right type that runs on the software platform where our tracing facility runs. We hope that our student will continue to make progress during his trip home, but his Internet connectivity has been sporadic, as feared.

June Accomplishments

4. Investigated several candidate attacks for testing.

July Planned Work

1. Choose one or more attacks for testing.
2. Perform preliminary testing.

Financial Data

1. Funds expended through June: $29,232
2. Percentage of total funds expended: 31%
3. Number of months remaining on contract: 7
5. Spending plan: On Track

FILE PROFILING FOR INSIDER THREATS

UCLA

July 2001

Contract F33615-00-C-1746 (LI 0001, Data Item A003)

Technical Point of Contact:
Peter Reiher
reiher@cs.ucla.edu

2. Introduction and Overview

This project is investigating the possibility of detecting the misbehavior of authorized users by analyzing patterns of file system access. Our thesis is that normal, well-behaved access is characteristically different than the forms of access used by an insider trying to step beyond his legitimate role. To do this, we must trace large quantities of legitimate access, derive models of appropriate behavior from those traces, and compare these models to sample misbehaviors.

The project is expected to 15 months, overall. We were fully funded at $94,154. Major team members are Dr. Peter Reiher, Dr. Geoff Kuenning, and graduate student Nam Nguyen.

2. Project Research Objectives

We must determine if normal access can be easily modeled across many users and across time. We must also determine if improper behavior actually does produce distinguishably different file system behavior.

3. Expected Project Results

If successful, our project will demonstrate that it is possible to detect important forms of insider misbehavior by intelligently observing file system behavior.

4. Achievements and Issues

Our lead graduate student, Nam Nguyen, is out of the country for the summer. His intent was to work remotely on the project, but circumstances have prevented that. Thus, our progress has not been along the same lines he had been examining. To avoid making no progress at all over the summer, we have been designing the architecture for the deployable system.

Progress on statement of work milestones:

1. Tracing program. We have gathered large quantities of file traces. Tracing continues on several machines in our office.

2. <u>Completion of first tracing cycle</u>. Completed. Models are under development. Our graduate student's summer travel will slow progress on this item until his return in the autumn.
3. <u>Development of code to support data analysis</u>. All code that we know is needed is completed. We expect periodically to find a need for more such code.
4. <u>Development of simple monitoring system</u>. Progress was made this month on this milestone, though we are still slightly behind schedule. Our efforts for the remainder of the summer will be focused on this item, both because we want to return it to schedule and because we can do the work in the absence of our graduate student. In essence, we believe we can use an existing piece of code to build much of this system. The Seer file hoarding facility already records file accesses and performs operations on them in real time. The requirements for this insider threat detection problem would be somewhat different, but not sufficiently so to require an entirely different piece of code. Essentially, the Seer system would need to feed file trace information to multiple implementations of file access models, each of which would then test the access against its model to determine if a threat is occurring. We would need further code to handle monitoring and signaling, and we would need to determine if we should add a sophisticated piece of multiplexing code to the system to decide which models would receive which file access records. We also need to examine the current state of the Seer code, since it has been adapted to other purposes in the past year, being altered substantially in the process. We will report further on this issue next month.
5. <u>Initial results from running the system live</u>. This work will start in the autumn. We expect results 2 to 3 months before the end of the contract, roughly in November or early December. Given that the period of performance was expanded to 15 months, this work is on schedule. This item cannot be met until item 4 is completed, so we are focusing more effort on that item.
6. <u>Final report</u>. This final report will be written when all other tasks have been completed.

6. Expenditures

July expenditures	Total: $8,131 Salary + related benefits and OH: $8012 Networking & supplies: $119
Total expended to date:	$37,367 (40% of funds)

Remaining funds are adequate to complete all work by contract end (5 Feb 02)

6. Program Status

Contract completion date: 5 February 2002
Percentage of contract period that has elapsed: 5.33%
Total funds that have been expended: $37,367
Percentage of contract funds that have been expended: 40%

FILE PROFILING FOR INSIDER THREATS

UCLA

August 2001

BAA-00-06-SNK; FRT Research Topic 2

Contract F33615-00-C-1746 (LI 0001, Data Item A003)

1 August – 31 August 2001

Technical Point of Contact:
Peter Reiher
reiher@cs.ucla.edu

3. Introduction and Overview

This project is investigating the possibility of detecting the misbehavior of authorized users by analyzing patterns of file system access. Our thesis is that normal, well-behaved access is characteristically different than the forms of access used by an insider trying to step beyond his legitimate role. To do this, we must trace large quantities of legitimate access, derive models of appropriate behavior from those traces, and compare these models to sample misbehaviors.

The project is expected to 15 months, overall. We were fully funded at $94,154. Major team members are Dr. Peter Reiher, Dr. Geoff Kuenning, and graduate student Nam Nguyen.

2. Project Research Objectives

We must determine if normal access can be easily modeled across many users and across time. We must also determine if improper behavior actually does produce distinguishably different file system behavior.

3. Expected Project Results

If successful, our project will demonstrate that it is possible to detect important forms of insider misbehavior by intelligently observing file system behavior.

4. Achievements and Issues

Our lead graduate student, Nam Nguyen, is out of the country for the summer. His intent was to work remotely on the project, but circumstances have prevented that. Thus, our progress has not been along the same lines he had been examining. As discussed in last month's report, to avoid making no progress at all over the summer, we have been designing the architecture for the deployable system. Another key project member, Geoff Kuenning, spent much of the last month on vacation; this slowed our progress further since he is the author of the existing code we intend to adapt for the deployable system. However, we were able to make some progress in this area before he departed.

Progress on statement of work milestones:

1. Tracing program. We have gathered large quantities of file traces. Tracing continues on several machines in our office.
2. Completion of first tracing cycle. Completed. Models are under development. Our graduate student's summer travel will slow progress on this item until his return in the autumn.

3. Development of code to support data analysis. All code that we know is needed is completed. We expect periodically to find a need for more such code.

4. Development of simple monitoring system. Some progress was made this month on this milestone. We have determined that the Seer system will indeed be suitable to adapt for this purpose. Our last report discussed briefly how Seer's file hoarding capability could be adapted to our purposes for this project. Both systems required real-time observation and analysis of a file access stream, though for different purposes. Seer is a somewhat old piece of code, but because portions of it have been used for a number of other purposes in our office, and because the original author is still a participant in this project, we have confidence that it will be workable. Relatively little effort will be required to make it work in our current operating system environment since it was originally designed for an earlier version of Linux. As much of Seer was kept out of the kernel as possible, making porting easier. Further, the major kernel hooks required by Seer are precisely those portions of the code that we have been using more or less continuously in our offices over the past three years, so they have already been ported to the version of Linux being used for this project. Porting the other portions of Seer will be relatively easy. Another major issue is that Seer only observed file opens and closes (and a few other operations, like changing the current working directory), not all reads and writes. Our tracing work has made it clear that we will need to look beyond opens and closes to detect misbehavior. While we already have altered the component of Seer that traps and records the other operations, we have not yet tried feeding these records into Seer's correlation facility. Some changes may be necessary there, and we will need to determine if Seer can keep up with the system with these extra records being fed in. A final, more fundamental difference is that we must entirely replace the code that Seer uses to analyze its data stream. Seer maintained huge tables of the order in which different files were accessed, but that is neither enough information nor the right information for detecting insider threats. All of this code will need to be changed.

5. Initial results from running the system live. This work will start in the autumn. We expect results 2 to 3 months before the end of the contract, roughly in November or early December. Given that the period of performance was expanded to 15 months, this work is on schedule. This item cannot be met until item 4 is completed, so we are focusing more effort on that item.

6. <u>Final report</u>. The final report will be written when all other tasks have been completed.

7. Expenditures

August expenditures	Total: $7656
	Salary + related benefits and OH: $7384
	Networking & supplies: $272
Total expended to date:	$45,022 (52% of funds)

Remaining funds are adequate to complete all work by contract end (5 Feb 02)

6. Program Status

Contract completion date: 5 February 2002
Percentage of contract period that has elapsed: 66.66%
Total funds that have been expended: $45,022
Percentage of contract funds that have been expended: 52%

FILE PROFILING FOR INSIDER THREATS

UCLA

September 2001

BAA-00-06-SNK; FRT Research Topic 2

Contract F33615-00-C-1746 (LI 0001, Data Item A003)

1 September – 30 September 2001

Technical Point of Contact:
Peter Reiher
reiher@cs.ucla.edu

4. Introduction and Overview

This project is investigating the possibility of detecting the misbehavior of authorized users by analyzing patterns of file system access. Our thesis is that normal, well-behaved access is characteristically different than the forms of access used by an insider trying to step beyond his legitimate role. To do this, we must trace large quantities of legitimate access, derive models of appropriate behavior from those traces, and compare these models to sample misbehaviors.

The project is expected to 15 months, overall. We were fully funded at $94,154. Major team members are Dr. Peter Reiher, Dr. Geoff Kuenning, and graduate student Nam Nguyen.

2. Project Research Objectives

We must determine if normal access can be easily modeled across many users and across time. We must also determine if improper behavior actually does produce distinguishably different file system behavior.

3. Expected Project Results

If successful, our project will demonstrate that it is possible to detect important forms of insider misbehavior by intelligently observing file system behavior.

4. Achievements and Issues

Our lead student returned from summer vacation, so we have been able to make more substantial progress. We have identified three candidate attacks that we will use to test our hypothesis that we can detect insider threats by watching normal and attack file access traces. These attacks are buffer overflow attacks that are intended to grant a normal user extra privileges. They operate by forcing a program that has such privileges (but ordinarily uses them in a limited and benign manner) to give the user unlimited access to the privileges. In Unix terms, a setuid root program with limited access to critical files is subverted to give the user a general-purpose execution shell with complete root privileges. The three attacks are on the crond (a utility that allows users to schedule jobs to run at particular times), KDE mail (a program that handles e-mail), and mars_new (an emulator that provides a subset of the functionality of Novell's file and print services for Linux). We have analyzed the normal behavior of these programs and the behaviors that are used to force the buffer overflows, and believe our system will detect the buffer overflow attacks. We will test that belief during the upcoming month.

Progress on statement of work milestones:

1. Tracing program. We have gathered large quantities of file traces. Tracing continues on several machines in our office.

2. Completion of first tracing cycle. Completed. Models are under development. Now that our graduate student has returned, progress on model development will speed up.

3. Development of code to support data analysis. All code that we know is needed is completed. We expect periodically to find a need for more such code.

4. Development of simple monitoring system. In the upcoming month, we will analyze the results of running the three programs discussed above in normal use versus their behavior when under buffer overflow attack. This information will feed back into the design of the monitoring system.

5. Initial results from running the system live. The results of our buffer overflow experiment will constitute one significant part of this deliverable. Work will continue on it during the autumn.

6. Final report. This final report will be written when all other tasks have been completed.

8. Expenditures

September expenditures	Total: $4660 Salary + related benefits and OH: $4423 Networking & supplies: $237
Total expended to date:	$49,682

Remaining funds are adequate to complete all work by contract end (5 Feb 02)

6. Program Status

Contract completion date: 5 February 2002
Percentage of contract period that has elapsed: 80%
Total funds that have been expended: $49,682
Percentage of contract funds that have been expended: 53%

FILE PROFILING FOR INSIDER THREATS

UCLA

October 2001

BAA-00-06-SNK; FRT Research Topic 2

Contract F33615-00-C-1746 (LI 0001, Data Item A003)

1 October – 31 October 2001

Technical Point of Contact:
Peter Reiher
reiher@cs.ucla.edu

5. Introduction and Overview

This project is investigating the possibility of detecting the misbehavior of authorized users by analyzing patterns of file system access. Our thesis is that normal, well-behaved access is characteristically different than the forms of access used by an insider trying to step beyond his legitimate role. To do this, we must trace large quantities of legitimate access, derive models of appropriate behavior from those traces, and compare these models to sample misbehaviors.

The project is expected to 15 months, overall. We were fully funded at $94,154. Major team members are Dr. Peter Reiher, Dr. Geoff Kuenning, and graduate student Nam Nguyen.

2. Project Research Objectives

We must determine if normal access can be easily modeled across many users and across time. We must also determine if improper behavior actually does produce distinguishably different file system behavior.

3. Expected Project Results

If successful, our project will demonstrate that it is possible to detect important forms of insider misbehavior by intelligently observing file system behavior.

4. Achievements and Issues

As mentioned in last month's report, we are working with three specific attacks to test our thesis: an attack on crond, an attack on KDE mail, and an attack on mars_new. These three vulnerabilities all arise from buffer overflow attacks, perhaps the most common method for a normal user to obtain root privileges on a Unix/Linux system.

Getting the vulnerabilities to work for testing purposes proved a bit more challenging than we expected. Typically, vulnerabilities depend on a particular version of the program in question, but also on a particular version of the operating system kernel (since often the program works differently, or not at all, on a different kernel version), and sometimes on versions of libraries or other system utilities. After some effort, we made all three vulnerabilities work on our test systems.

We then began to test methods of automatically detecting the attack. In addition to keeping file traces, we also keep information about which processes forked and executed other processes, so we are also using that data to detect these attacks. Based on the entire set of data available and our observation of how attack tools that attempt to use these vulnerabilities behave, we have observed at least four opportunities to automatically detect attempts to use the vulnerabilities under examination:

1. Examining what other users are doing by attempting to access files belonging to other users in the /proc area. (This area contains files that represent running processes.)

2. Attempting to access files for which the user does not have read, write, or execute permission. This behavior frequently indicates an attempt to find mistakes by the system administrator in setting these permissions.

3. Frequently changing the working directory of a process. This behavior sometimes indicates an attempt to place files in directories that are owned by privileged users or have special treatment. For example, if the attacker wants to place a Trojan Horse program in a place where the root user is likely to encounter it, the attacker is likely to check many possible directories before finding a vulnerable one.

4. Passing long arguments to programs, especially long arguments that contain machine instruction codes. This is characteristic of most buffer overflow attacks.

We will further investigate these and other attack patterns in the upcoming month.

Progress on statement of work milestones:

1. Tracing program. We have gathered large quantities of file traces. Tracing continues on several machines in our office.

2. Completion of first tracing cycle. Completed. Models have been developed.

3. Development of code to support data analysis. All code that we know is needed is completed. We expect periodically to find a need for more such code.

4. Development of simple monitoring system. We are using the information described above to design this system.

5. Initial results from running the system live. The results above are part of this milestone. Work will continue on it during the autumn.

6. Final report. This final report will be written when all other tasks have been completed.

9. Expenditures

October expenditures	Total: $9970 Salary + related benefits and OH: $8808 Networking & Supplies: $1162
Total expended to date:	$59,652

Remaining funds are adequate to complete all work by contract end (5 Feb 02)

6. Program Status

Contract completion date: 5 February 2002
Percentage of contract period that has elapsed: 80%
Total funds that have been expended: $59,652
Percentage of contract funds that have been expended: 63.35%

FILE PROFILING FOR INSIDER THREATS

UCLA

November 2001

BAA-00-06-SNK; FRT Research Topic 2

Contract F33615-00-C-1746 (LI 0001, Data Item A003)

1 November – 30 November 2001

Technical Point of Contact:
Peter Reiher
reiher@cs.ucla.edu

6. Introduction and Overview

This project is investigating the possibility of detecting the misbehavior of authorized users by analyzing patterns of file system access. Our thesis is that normal, well-behaved access is characteristically different than the forms of access used by an insider trying to step beyond his legitimate role. To do this, we must trace large quantities of legitimate access, derive models of appropriate behavior from those traces, and compare these models to sample misbehaviors.

The project is expected to 15 months, overall. We were fully funded at $94,154. Major team members are Dr. Peter Reiher, Dr. Geoff Kuenning, and graduate student Nam Nguyen.

2. Project Research Objectives

We must determine if normal access can be easily modeled across many users and across time. We must also determine if improper behavior actually does produce distinguishably different file system behavior.

3. Expected Project Results

If successful, our project will demonstrate that it is possible to detect important forms of insider misbehavior by intelligently observing file system behavior.

4. Achievements and Issues

As discussed in last month's report, we are working with three specific attacks to test our thesis. These are an attack on crond, an attack on KDE mail, and an attack on mars_new. These three vulnerabilities all arise from buffer overflow attacks, perhaps the most common method for a normal user to obtain root privileges on a Unix/Linux system.

We have overcome the challenges mentioned in last month's report and were actually able to test these exploits. We ran them against our altered tracing kernel to determine their file access patterns, which should allow us to distinguish their file behavior from benign use of the same programs. Getting to this stage required some porting and some changes to our code. For example, previously we had not traced root file accesses, since doing so could sometimes cause a deadlock condition, and our earlier uses of the tracing code did not require root access. Having determined that we need root tracing for insider threat detection purposes, we changed the code to do such tracing and avoid the deadlock.

Our original plan was to compare patterns of which files were opened in which order for normal use of a program to the same information for abnormal use. After examining traces of attack behavior, we found easier ways to use file trace information or slight additions to traced information to detect the attacks. For example, one can detect a probable buffer overflow attack by catching system calls and examining their arguments for unusually long arguments. Many buffer overflow attacks require that the program make a system call with the overflow information in order to clobber the stack, so this method catches the attack more easily than a general pattern match of file activity.

Similarly, one of the attacks that involves linking to other users' files can be more easily detected by an attempt to create a symbolic link to a file owned by another user when the attacker does not have read/write/execute permissions for that file. Such links are legal to create, but are most typically used to fool a program into using a file it should not use. Again, a simple check for a behavior that, while legal, is never used by proper programs has proved easier than a generic pattern match.

While these simple tests are effective against these attacks, they are ad hoc responses to particular attacks. We hope to develop enough experience with various attacks and detection methods to generalize these specific signatures into a model usable for many types of attacks.

Progress on statement of work milestones:

1. Tracing program. We have gathered large quantities of file traces. Tracing continues on several machines in our office. We updated the tracing code to also trace root activity.

2. Completion of first tracing cycle. Completed. Models have been developed.

3. Development of code to support data analysis. All code that we know is needed is completed. We expect periodically to find a need for more such code.

4. Development of simple monitoring system. We are using the information described above to design this system.

5. Initial results from running the system live. The results above are part of this milestone. Work will continue on it during the winter.

6. Final report. This final report will be written when all other tasks have been completed.

10. Expenditures

November expenditures	Total: $4887
	Salary + related benefits and OH: $4887
	Networking & supplies: $0
Total expended to date:	$64,539 (69% of funds)

Remaining funds are adequate to complete all work by contract end (5 Feb 02)

6. Program Status

Contract completion date: 5 February 2002
Percentage of contract period that has elapsed: 86%
Total funds that have been expended: $64,539
Percentage of contract funds that have been expended: 69%

FILE PROFILING FOR INSIDER THREATS

UCLA
BAA-00-06-SNK; FRT Research Topic 2
Contract F33615-00-C-1746 (LI 0001, Data Item A003)
1 January – 31 January 2002

Technical Point of Contact:
Peter Reiher
reiher@cs.ucla.edu

7. Introduction and Overview

This project is investigating the possibility of detecting the misbehavior of authorized users by analyzing patterns of file system access. Our thesis is that normal, well-behaved access is characteristically different than the forms of access used by an insider trying to step beyond his legitimate role. To do this, we must trace large quantities of legitimate access, derive models of appropriate behavior from those traces, and compare these models to sample misbehaviors.

The project is expected to 15 months, overall. We were fully funded at $94,154. Major team members are Dr. Peter Reiher, Dr. Geoff Kuenning, and graduate student Nam Nguyen.

2. Project Research Objectives

We must determine if normal access can be easily modeled across many users and across time. We must also determine if improper behavior actually does produce distinguishably different file system behavior.

3. Expected Project Results

If successful, our project will demonstrate that it is possible to detect important forms of insider misbehavior by intelligently observing file system behavior.

4. Achievements and Issues

This was the last full month of work on this project. We concentrated on wrapping up incomplete details and studying what we had learned from our efforts. The complete lessons will be described in our final report, but generally we learned that examination of file access traces can indeed often detect insider misbehavior. Our original thoughts on how to use this information did not always prove fruitful. However, we discovered several other methods of using the information, and also discovered that some other easily gathered information can make detection of misbehavior easier and more accurate. For example, pure use of file access information is not as powerful as also including information about process execution trees.

We did not complete two of our original milestones because we determined their purpose was better served by a different approach. We do not have a monitoring system that can be used in live operations, and, as a result, also do not have results of live performance of the system. The purpose of these milestones was to demonstrate that the system could detect actual attempts by insiders to misbehave. We determined that we could demonstrate this better by running actual attack tools and methods and demonstrating how they resulted in behavior well outside our models of normal behavior. Actually building the system to detect misbehavior in live operation would have required much engineering unrelated to the basic research goal of our project. We used the time instead to perform closer analysis of our trace data than originally planned and to develop a wider array of models.

Progress on statement of work milestones:

7. Tracing program. Completed.
8. Completion of first tracing cycle. Completed.
9. Development of code to support data analysis. Completed.
10. Development of simple monitoring system. Replaced by live testing of real attack tools and methods.
11. Initial results from running the system live. Replaced by live testing of real attack tools and methods.
12. Final report. In progress. To be delivered in late February.

11. Expenditures

January expenditures	Total: $17,429 Salary + related benefits and OH: $17,057 Networking & supplies: $372
Total expended to date:	$94,154 (100% of funds)

6. Program Status

Contract completion date: 5 February 2002
Percentage of contract period that has elapsed: 100%
Total funds that have been expended: $94,154
Percentage of contract funds that have been expended: 100%